Pedro Miguel Coelho Estevão

Licenciado em Ciências da Engenharia
Eletrotécnica e de Computadores

Sistema de aquisição e processamento de imagem com computador e FPGA

Dissertação para obtenção do Grau de Mestre em
Engenharia Eletrotécnica e de Computadores

Orientador: Doutor Filipe de Carvalho Moutinho, Professor Auxiliar,
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente: Prof. Doutora Anabela Monteiro Gonçalves Pronto
Arguente: Prof. Doutor Luís Filipe dos Santos Gomes
Vogal: Prof. Doutor Filipe de Carvalho Moutinho

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2020

Sistema de aquisição e processamento de imagem com computador e FPGA

Agradecimentos

Gostaria de iniciar os meus agradecimentos ao meu orientador, o Professor Filipe de Carvalho Moutinho, por me ter dado a oportunidade de realizar este projeto, por me ter disponibilizado tempo e dedicação para me apoiar ao longo da realização desta dissertação.

De seguida agradeço à Faculdade de Ciências e Tecnologias, instituição onde realizei o meu percurso formativo e ao Departamento de Engenharia Eletrotécnica e de Computadores pelo todo o conhecimento partilhado e desafios propostos que contribuíram para a minha formação académica e pessoal.

Queria também fazer um agradecimento aos meus colegas de curso que me acompanharam ao longo destes anos de persistência e de pressão, que me acompanharam em trabalhos de grupo e noites de estudo e que também me marcaram com momentos, dentro e fora da faculdade que irei guardar para o resto da vida. Agradeço a Miguel Lopes, Nuno Carvalhão, Diogo Lograd o, Manuel Faustino, Carlos Diogo, João Ramos, João Bento, Marco Albuquerque e Victor Fernandes por terem estado ao meu lado nesta fase da minha vida.

Um especial agradecimento à minha família por todo o apoio contínuo em todos os momentos, por me puxaram para sempre melhorar em todos os aspetos e por nunca desistir daquilo que desejo.

Agradeço carinhosamente à minha namorada Ana Reis, por estudar comigo, por realizar trabalhos a meu lado e acreditar no meu sucesso, nos bons e maus momentos.

A todos, um enorme obrigado.

Resumo

O processamento de imagem é um conceito muito vasto e utilizado em diversas áreas, tais como diagnóstico e tratamento médico, aplicação industrial, análise meteorológica, entre outras. Há algumas décadas o processamento de imagem era realizado maioritariamente de forma analógica. No entanto, com a gradual evolução tecnológica e a capacidade de processamento dos computadores, as técnicas de processamento foram aos poucos sendo substituídas por métodos digitais. Bibliotecas de processamento de imagem, das quais se destacam a OpenCV, são amplamente utilizadas, facilitando o desenvolvimento de sistemas de processamento de imagem.

As FPGA (*Field-Programmable Gate Array*) são dispositivos reconfiguráveis que podem ser utilizados para implementar circuitos digitais. Embora sejam frequentemente referidas vantagens das FPGA, em certas situações quando comparadas a computadores, a sua utilização pode ter uma menor performance, nomeadamente em sistemas de processamento de imagem. Isto pode ser explicado pela complexidade da tecnologia e pela complexidade dos ambientes de desenvolvimento.

Tendo em conta que o objetivo deste trabalho é contribuir para a utilização de FPGA na aquisição e processamento de imagens, neste documento propõe-se um sistema de aquisição e processamento de imagem composto por FPGA e computador. A FPGA é utilizada para fazer a aquisição e o pré-processamento de imagem, enquanto o computador é utilizado para o restante processamento. É proposta a utilização de um computador de baixo custo, um Raspberry Pi, de fácil utilização e que permite a utilização de OpenCV. Para a comunicação entre a FPGA e o computador é utilizado o protocolo SPI (*Serial Peripheral Interface*).

Palavras-chave: Processamento de imagem, Plataformas Heterogéneas, *Field Programmable Gate Array* (FPGA), VHDL, *Hardware*, *Software*, Microprocessador

Abstract

Image processing is a very broad concept in several areas. It is used in areas such as medical treatment, industrial applications, meteorological analysis, among many others. A few decades ago, image processing was performed mostly in an analog way. But with the gradual technological evolution and the speed capability of computers getting better and better every day, processing techniques were gradually being replaced by digital methods. Image processing libraries, which OpenCV stands out, are widely used, facilitating the development of image processing systems.

FPGA (Field-Programmable Gate Array) are reconfigurable devices that can be used for digital circuits. Although the advantages of FPGA are often mentioned, when compared to computers, in some cases their use have less performance, namely in image processing systems. This can be explained by the fact of their complexity of the technology and the complexity of the development environments.

Keeping in mind that the objective of this work is to contribute to the use of FPGA in the acquisition and processing images, in this document it is proposed an image acquisition and processing system composed of a FPGA and computer. The FPGA is utilized for image acquisition and pre-processing, while the computer is used for the rest of the processing work. It is proposed to use a low-cost computer, a Raspberry Pi, which is easy to use and allows the utilization of OpenCV. For the communication between the FPGA and the computer is used the SPI (Serial Peripheral Interface) protocol.

Keywords: Image Processing, Heterogeneous Platforms, FPGA, VHDL, Hardware, Software, Microprocessor

Conteúdo

Índice de Figuras

Índice de Tabelas

Lista de Siglas

ACK	*Acknowledgement*
ASIC	*Application Specific Integrated Circuit*
AXI	*Advanced eXtensible Interface*
CLB	*Configurable Logic Block*
CPLD	*Complex Programmable Logic Device*
FPGA	*Field-Programmable Gate Array*
GPIO	*General-Purpose Input/Output*
HDL	*Hardware Description Language*
HLS	*High-Level Synthesis*
I2C	*Inter-Integrated Circuit*
IEEE	*Institute of Electrical and Electronics Engineers*
IOB	*Input/Output Block*
LSB	*Least Significant Bit*
LUT	*Look Up Table*
MATLAB	*MATrix LABoratory*
MSB	*Most Significant Bit*
NACK	*Negative Acknowledgement*
RGB	*Red*, *Green*, *Blue*
SCL	*Serial Clock*
SD	*Secure Digital*

SDA *Serial Data*

SDK *Software Development Kit*

SoC *System on Chip*

SPI *Serial Peripheral Interface*

VHDL *VHSIC Hardware Description Language*

VHSIC *Very-High-Speed Integrated Circuit*

1

1. Introdução

Este capítulo introduz o trabalho realizado nesta da dissertação, que ronda à volta da utilização de FPGA (*Field-Programmable Gate Array*) para processamento de imagem. Este capítulo apresenta as motivações para a realização deste trabalho, bem como os seus objetivos e contribuições. Finalmente, neste capítulo também se apresenta a estrutura e como foi organizada a dissertação.

1.1Motivação

Habitualmente o processamento de imagem é realizado de forma digital, como tal as imagens são manipuladas por computadores. Existem vários ambientes de desenvolvimento e bibliotecas para realizar este trabalho, mas sem dúvida uma das mais populares é a OpenCV. A biblioteca de *software Open Source Computer Vision* (OpenCV) tem um conjunto de funções que fornece ao utilizador uma simples infraestrutura com diversas aplicações ao nível de operações relacionadas com imagens.

Mas nem sempre o processamento é feito em *software*, este também pode ser realizado em *hardware* e como tudo, tem as suas vantagens e desvantagens. Existe ainda a possibilidade de se realizar um processamento misto, utilizando plataformas heterogéneas, e assim sendo, poderá ser possível ter o melhor dos dois mundos. Podemos ter a facilidade de implementação para algoritmos mais complexos (em *software*), e por outro lado a rapidez de execução para algoritmos mais acessíveis, capazes de implementar em *hardware*.

1.2 Objetivos e Contribuições

O principal objetivo deste projeto foi desenvolver um sistema de aquisição e processamento de imagem com FPGA e computador para realizar processamento em tempo real. Sendo uma parte implementado em *hardware* e outra parte em *software*.

Nesta tese realizou-se um levantamento de diversas plataformas de processamento de imagem com o intuito de perceber quais as melhores opções a tomar em relação ao equipamento a usar consoante a finalidade do projeto, qual o protocolo de comunicação a usar para envio de informação entre o *software* e o *hardware* e que partes implementar em cada componente.

Nesta tese pretendeu-se desenvolver um sistema de aquisição e processamento de imagem com FPGA e computador para realizar processamento em tempo real. Sendo uma parte implementado em *hardware* e outra parte em *software*.

Este sistema terá o objetivo de ser uma plataforma heterogénea de prototipagem e/ou aprendizagem, acessível a pessoas que não tenham um vasto conhecimento em desenvolvimento de circuitos, programação e experiência associada a processamento de imagem. Espera-se que este projeto dê a essas pessoas a oportunidade de a usarem e a alterarem de acordo com os seus objetivos.

Neste sentido, o sistema desenvolvido, composto por câmara, FPGA e computador, tem de capturar imagens através de uma câmara e as transmitir para a FPGA através de uma interface. Após este recolher de informação pode então ser feito um pré-processamento em FPGA. Posteriormente, a FPGA transmite dados a um computador (um Raspberry Pi) para que este os possa processar utilizando bibliotecas de *software*, nomeadamente OpenCV. Para a comunicação entre a FPGA e o computador usou-se o protocolo de comunicação SPI (*Serial Peripheral Interface*) que permite que esta transmissão de dados possa ser feita ser quaisquer problemas ou erros.

A FPGA, para além de receber as imagens da câmara e de as transmitir para o computador, também aplica um conjunto de métodos de pré-processamento de imagem, de

acordo com o solicitado pelo computador. Alguns dos métodos implementados e disponibilizados pela FPGA são: conversão para escala de cinzentos, conversão para preto e branco, alteração de brilho e alteração de contraste.

1.3 Estrutura da tese

Além deste primeiro capítulo que serve de introdução, a dissertação tem a seguinte estrutura.

No segundo capítulo, descreve-se o estado de arte, discute-se o processamento de imagem, quais as melhores medidas a tomar no que toca à velocidade de execução, a complexidade de implementação, quais os componentes de *software* e *hardware* mais vantajosos para o problema em questão, quais os métodos de comunicação entre estes e para complementar todas estas características estão descritos trabalhos, questões e conclusões de investigadores com desafios acerca de processamento de imagem.

No terceiro capítulo, descreve-se o sistema desenvolvido. Depois de uma introdução é apresentado o *hardware* utilizado, as ligações entre a FPGA, o Raspberry e a câmara, o módulo SPI *slave* implementado na FPGA, o funcionamento do sistema e a sua arquitetura.

O quarto capítulo apresentam-se os testes e resultados. Adicionalmente, descreve-se o módulo SPI implementado no Raspberry e que suportou os testes efetuados. Por fim apresentam-se os recursos utilizados da FPGA.

No quinto capítulo, realiza-se uma apreciação acerca das conclusões retiradas acerca desta plataforma heterogénea e da comunicação entre os seus componentes e perspetiva-se alguns trabalhos futuros que se podem basear neste trabalho.

2

2. Estado de Arte

Neste capítulo apresenta-se a pesquisa efetuada para esta dissertação, os conceitos vistos e explicados de modo a compreender o contexto por detrás do processamento de imagem.

Na secção 2.1 é explicado o fundamento do processamento de imagem, porque é importante na atualidade, em que áreas é utilizado e qual a sua função.

Na secção 2.2 são referidas duas tecnologias mais conhecidas que presentemente são usadas para a prática de processamento de imagem, juntamente com as suas bibliotecas e funções para tal. Serão descritos exemplos em que usaram estas tecnologias e depois é realizada uma comparação entre ambas.

Na secção 2.3 fala-se acerca das FPGA, qual o seu contributo para este tipo de projetos de processamento de imagens e de linguagens de descrição de *hardware*. É ainda elaborado sobre o futuro das FPGA.

Na secção 2.4 referem-se algumas plataformas heterogéneas, enquanto que na secção 2.5 são apresentadas interfaces de comunicação que suportam a comunicação entre *hardware* e *software*.

2.1 Processamento de imagem

O processamento de imagem é a manipulação de imagens, pois estas representam e guardam informação. Processamento de imagem nem sempre está relacionado com a imagem na sua totalidade, porque por vezes, apenas é necessário retirar informação de uma fração da imagem. Esta técnica é essencial em diversas áreas, tais como, medicina, agricultura, robótica, militar, entre outras. É de notar que em situações que envolvam sensores e sistemas de segurança, o processamento de imagem é indispensável. Atualmente, qualquer sistema que inclua uma câmara necessita de algum tipo de processamento.

Áreas que utilizam processamento de imagem

Ciência Forense – As técnicas mais comuns nesta área são de reconhecimento, correspondência de padrões, segurança e propósitos biométricos. A ciência forense é baseada na informação sobre indivíduos, assim sendo, muitas vezes os dados de entrada estão relacionados com impressões digitais, olhos, faces, imagens que definem uma pessoa. [1]

Medicina – Desde a descoberta dos raios X, estes têm sido usados para representar por imagens partes do corpo humano para propósitos de diagnósticos. Estas imagens necessitam de manipulações e filtros, como sobras, brilhos, aumento ou diminuição de brancos ou pretos, para conseguir identificar com precisão de o paciente tem algum osso partido, ou algum problema no cérebro ou na mama. Depende da doença que se pretende diagnosticar, a maneira como se modifica a imagem é diferente. [1][2]

Agricultura – Processamento de imagem também pode ser usado para detetar uma de uma planta que esteja doente, ao analisar uma coleção de imagens de essa planta. Tradicionalmente, seria um especializado em plantas para verificar caso haja algum problema, mas isso custaria muito dinheiro e um grande consumo de tempo. Assim o próprio agricultor consegue analisar as imagens e caso verificar se naquele período a planta se encontra doente, então pode guardar as imagens para uma referência no futuro. [3]

Manufatura – Também é bastante usado nos sistemas robóticos e de uma forma muito crucial, uma vez que o robô necessita de decidir o que fazer no momento e de se posicionar no seu ambiente. Há um exemplo de um robô que reconhecia uma pessoa através de uma câmara e calculava a distância até esta mesma para evitar obstáculos e usado processamento de imagem no sentido de identificar a pessoa de outros objetos. O robô tinha como objetivo seguir a pessoa num ambiente fechado. [4]

2.2 Tecnologias usadas para processamento de imagem

Nesta secção vamos abordar as tecnologias que habitualmente se utilizam para realizar o processamento de imagem e fazer uma comparação entre as diferentes abordagens para definir vantagens e desvantagens na relação entre elas.

2.2.1 MATLAB

MATLAB (*MATrix LABoratory*) [5] é uma ferramenta de *software* de análise de dados, de visualização que suporta um diverso leque de funções para cálculo numérico e operação de matrizes. Inclui excelentes capacidades gráficas, juntamente com um ambiente de desenvolvimento derivado de uma linguagem de programação de alto nível.

O MATLAB tem sido bastante reconhecido ao longo do tempo, pelos cientistas, engenheiros e investigadores que o utilizam, por muitos benefícios que transmite, entre eles, a sua enorme capacidade de funções especializadas. MATLAB é utilizado em diversas áreas, desde questões financeiras, redes neuronais a processamento de imagem.

O MATLAB tem uma série de simples funções que ajudam na operação de processamento de imagem, como estas apresentadas na tabela 2.1.

Tabela 2.1: Algumas funções que o MATLAB disponibiliza para processar imagens (adaptado de [2]).

Imread	Permite ao utilizador ler um ficheiro de imagem de qualquer tipo
rgb2gray	Realiza uma conversão do rgb para o seu equivalente numa escala de cinzento
image	Apresenta a imagem no mapa de cor pretendido
impixel	Retorna o valor do rgb do pixel em questão
imwrite	Permite que a imagem seja guardada numa extensão à escolha do utilizador

O processamento de imagem tem um vasto grupo de técnicas e algoritmos que facilmente se consegue manifestar em MATLAB, entre a remoção de ruído, *sharpening*, *deblurring*, extração de cantos, binarização, *blurring* e segmentação de objetos (exemplo mostrado na figura 2.1), entre outras operações.

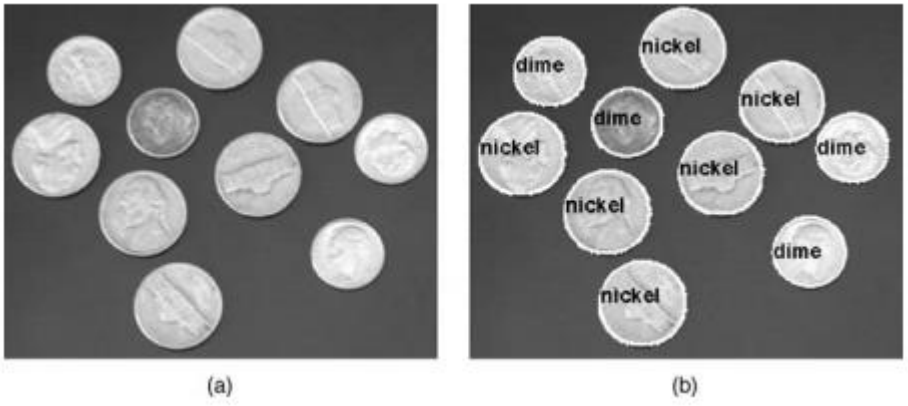

Figura 2.1: Segmentação dos objetos e classificação: a) imagem original; b) imagem após a segmentação e classificação das moedas. [6]

Posto isto, conseguimos compreender que o MATLAB é uma ferramenta muito útil para realizar o processamento digital de imagens, mas não é o único *software* capaz destas operações. Existe também uma outra ferramenta, bastante conhecida, denominada de OpenCV. Também é capaz de realizar as mesmas operações que o MATLAB. Na figura 2.2 está representado um gráfico que faz uma comparação do tempo que demora o MATLAB e o OpenCV a fazer um filtro de Sobel numa imagem. Consegue-se analisar que o OpenCV é bastante mais rápido na sua execução.

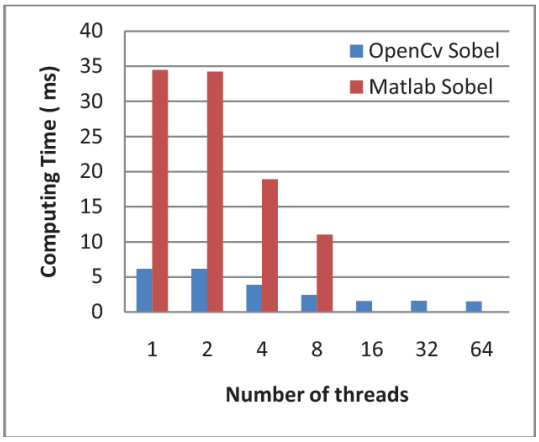

Figura 2.2: Comparação entre um filtro de Sobel usando o OpenCV e Matlab. [7]

2.2.2 OpenCV

A OpenCV (*Open Source Computer Vision Library*) [8] foi originalmente desenvolvida pela Intel, em 2000, e é uma biblioteca de *software* que visa a desenvolver aplicações na área de visão computacional e como tal também permite realizar processamento de imagem com bastante facilidade. Esta biblioteca foi desenvolvida para que os algoritmos fossem implementados em C++. Porém, também dá suporte a diferentes linguagens tais como Java, Python e Visual Basic.

A OpenCV tem funções idênticas às do MATLAB que foram descritas anteriormente e para além disso tem também filtros e deteção de objetos. A figura 2.3 mostra dois filtros semelhantes

que têm como objetivo a deteção de contorno de objetos, utilizando um algoritmo que aplica um filtro de Sobel.

Figura 2.3: Aplicação de vários diversos filtros de imagens distintas em OpenCV. Primeira coluna: Imagem original. Segunda coluna: Aplicação do *canny edge detection*. Terceira coluna: *COM edge detection*. [9]

Para concluir podemos afirmar que o OpenCV é uma ótima plataforma para realizar processamento de imagem, pois foi desenhada com esse mesmo propósito. Sabemos que o OpenCV e MATLAB são dois *softwares* bastante acessíveis com muito conteúdo para ajudar aos seus utilizadores e bastante intuitivos de desenvolver aplicações de processamento de imagem.

2.3 FPGA

Vamos agora discutir o valor de processamento de imagem realizado em *hardware*. A FPGA é um circuito integrado que pode ser reconfigurado, o permite que as suas funcionalidades

possam ser definidas pelos seus utilizadores e não só pelos seus fabricantes. As FPGA eram usadas no passado para aplicações de baixo volume, complexidade e volume, mas hoje em dia ultrapassam a barreira dos 500MHz [10].

A FPGA é constituída por blocos de entrada e saída (*Input/Output Block IOB*), blocos lógicos configuráveis (CLB) e interconexões entre estes blocos [10]. Os blocos lógicos implementam as funções lógicas e contam com vários componentes, tais como Look-Up Table's (LUT's), Flip Flop's e Multiplexers. Os blocos lógicos funcionam separadamente logo conseguem operar em paralelo, e estes são reconfiguráveis, logo podemos programar as suas conexões e juntá-los para criar algo com mais significado. Os blocos de entrada e saída tem a função de realizar a interface entre os blocos lógicos e componentes externos caso haja na arquitetura do projeto.

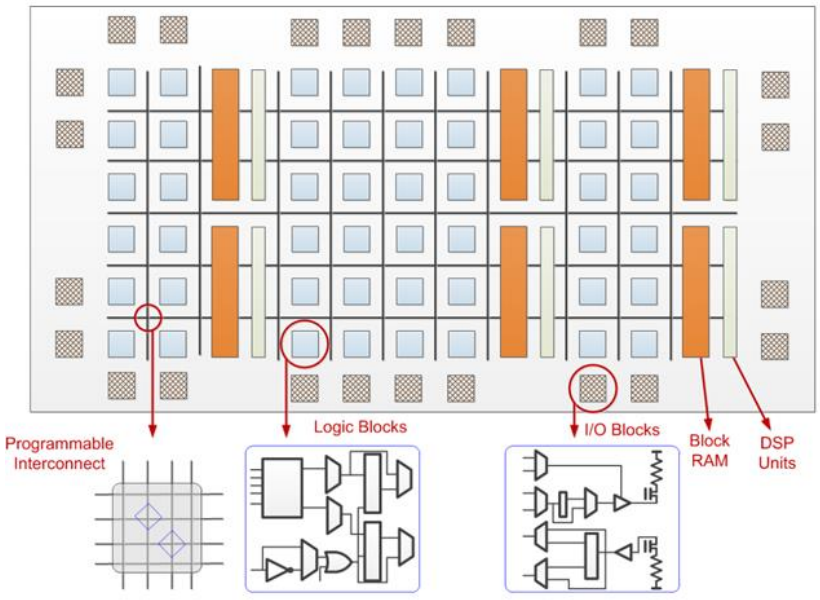

Figura 2.4: Arquitetura de uma FPGA. [10]

2.3.1 O uso de FPGA em projetos de processamento de imagem

Têm sido diversos os projetos de processamento de imagem em que as FPGA têm sido utilizadas. Muitos destes projetos comparam as FPGA com outras plataformas, nomeadamente CPU (*Central Process Unit*) e GPU (*Graphics Processing Unit*) [11][12][13][14].

Um destes projetos [11] teve como objetivo a comparação entre CPU e FPGA num sistema que tem como função detetar rachas de paredes. Foi usada um PicoZed Embedded Vision Kit como *hardware* e foi usado o Vivado HLS (*High-Level Synthesis*) [15] como ambiente de desenvolvimento. Foi utilizado um cartão SD (*Secure Digital*) para guardar a imagem. Para comparar implementou-se também o algoritmo em *software*, em MATLAB, num comutador convencional. Os resultados foram que a FPGA teve melhores resultados ao nível da performance, mas essencialmente na questão da eficiência energética, pois teve uma menor *footprint* e isto tudo tendo a mesma capacidade de precisão para definir a falha na parede.

2.3.2 Linguagens de Descrição de Hardware

Existem vários tipos de linguagem de descrição de *hardware*, sendo que as mais comuns entre os utilizadores são o VHDL e o Verilog. Neste projeto iremos utilizar a linguagem VHDL, por isso em baixo segue-se uma secção descrevendo um pouco a origem dessa linguagem.

2.3.2.1 VHSIC Hardware Description Language

Linguagem de descrição de *hardware* VHSIC "***V**ery **H**igh **S**peed **I**ntegrated **C**ircuits*" *Hardware Description Language* (VHDL), é uma linguagem usada para facilitar o design (projeto/concepção) de circuitos digitais em CPLD (*Complex Programmable Logic Device*), FPGA e ASIC (*Application Specific Integrated Circuit*).

A linguagem VHDL foi desenvolvida pelo Departamento de Defesa americano na década de 1980 para documentar o comportamento de ASIC [16]. Pretendia-se desenvolver circuitos através da descrição de algoritmos de forma textual, sem recurso a esquemáticos. A linguagem VHDL permite não só a documentação e descrição de circuitos, mas também suporta a síntese, simulação e os testes. O VHDL é atualmente uma norma do IEEE (*Institute of Electrical and Electronics Engineers*) [17].

2.3.2.2 Verilog

O Verilog foi uma das primeiras linguagens de descrição de *hardware*. Também criada na década de 1980, foi normalizada pelo IEEE em 1995 [18]. Uma comparação entre as linguagens Verilog e VHDL é apresentada em [19]. Em termos de operadores ambas as linguagens têm os mesmos. O Verilog pode ser um pouco mais fácil de compreender para quem não tem qualquer experiência em programação de *hardware*, isto também porque o VHDL é menos intuitivo. O VHDL tem que bibliotecas que facilitam a integração de um projeto mais vasto e complexo, já o Verilog não tem esse conceito de bibliotecas devido à sua origem como uma linguagem interpretativa. O VHDL tem maior facilidade na reutilização de funções e procedimentos, devido ao facto de os poder incluir em *packages*. O mesmo não se aplica ao Verilog, pois não existe conceção de *packages*, as funções necessitam de estar dentro dos próprios módulos. Em termos de tipo de dados, o VHDL tem uma enorme magnitude, isto significa que proporciona funções para fazer a conversão entre tipos de dados. Comparando ao VHDL, os tipos de dados do Verilog são bastante mais simples, os dados não são criados pelo utilizador, mas sim pela própria linguagem Verilog, como o exemplo de *wire* para variáveis associados a ligações de dados ou *reg* para o mesmo associado a registos. De forma geral, é mais benéfico usar a linguagem VHDL do que Verilog para projetos de maior nível e complexidade. Um estudo que compara estas duas linguagens conclui que comparando a capacidade de modelação, o VHDL e o Verilog cobrem diferentes zonas do espectro em relação aos níveis do comportamento de abstração.

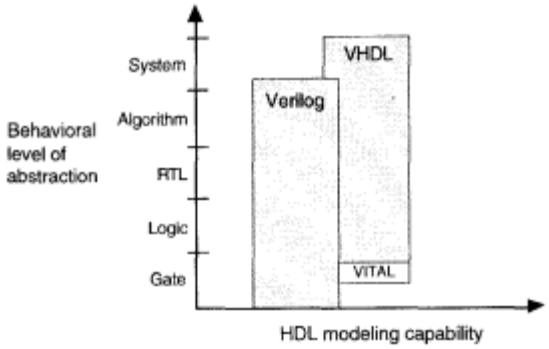

Figura 2.5: Capacidade de modelação HDL. [19]

2.3.3 Qual o futuro das FPGA?

Cada vez mais as indústrias tendem a incluir as FPGA como parte do paradigma das plataformas heterogéneas, em que se refere a um sistema que utiliza mais do que um tipo de circuito para realizar processamento. A FPGA disponibiliza um poder computacional bastante elevado e com uma boa relação custo-eficiência e que a torna adequada para projetos de rápida prototipagem e de baixa complexidade de implementação. Existem casos em que a FPGA tem uma melhor performance que o computador sobretudo em níveis de eficiência energética. Depende de várias condições claro, mas em situações de aplicações em tempo real, a FPGA tende a ser mais adaptável.[10][20]

2.4 Plataformas Heterogéneas

Tendo em conta o que foi referido anteriormente, podemos contextualizar e perceber que ambas as implementações em *software* e *hardware* têm as suas vantagens e desvantagens. Então uma boa solução será juntar as duas componentes retirando o melhor partido de cada uma delas. Podendo então realizar um pré-processamento em FPGA, algoritmos mais acessíveis de

implementar e tendo um tempo de execução menor e posteriormente transmitir a informação para o processador de modo a finalizar a implementação, realizando a parte mais complexa de desenvolver.

Posto isto, os fabricantes de FPGA, como por exemplo a Xillinx [21] e a Intel [22], criaram circuitos integrados que são SoC (*System on Chip*). Estes sistemas têm incorporado FPGA e processador, o que implica logo maior simplicidade ao projeto, pois apenas temos um único dispositivo. Um desses casos é a placa Zybo Z7: Zynq-7000 ARM/FPGA SoC *Development Board* [23]. Como o nome indica esta placa da Diligent tem uma arquitetura SoC, pois integra um processador Dual-Core ARM Cortex-A9 com uma lógica Xillinx 7-series FPGA. A componente em *software* é desenvolvida no Vivado Design Suite & Xillinx SDK.

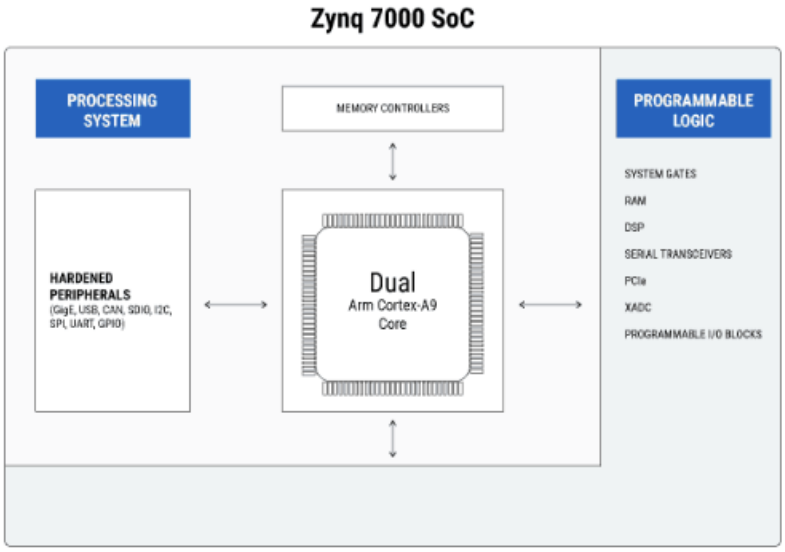

Figura 2.6: Estrutura da placa Zynq 7000 SoC. [24]

Existe também um multiprocessador System-On-Chip (MPSoC) que inclui múltiplos cores que também é utilizado para desenvolvimento de plataformas heterogéneas, e tem a vantagem que comunicar e partilhar informação entre os processadores. Este tipo de aplicação tem aumentado o seu interesse na área de sistemas robóticos, devido às suas capacidades de performance. [25]

Podendo ter a hipótese de ter as duas componentes, *software* e *hardware* em separado, como por exemplo, uma Xilinx Spartan-6 FPGA a fazer o pré-processamento e um Raspeberry Pi para fazer o restante processamento. Daqui se percebe que irá ser necessário existir comunicação entre as duas componentes. Tal como neste trabalho, são diversos os trabalhos com ênfase na utilização de FPGA e Raspberry Pi [26][27][28].

2.5 Protocolos de comunicação

O protocolo mais usado neste tipo de plataformas SoC da Xilinx é o *Advanced eXtensible Interface* (AXI). Este protocolo é uma interface síncrona, capaz de altas frequências e consegue realizar comunicação entre vários *masters* e *slaves*. O AXI foi introduzido em 2003 e tem diversas variações tais como, AXI3, AXI4, AXI4-Lite, AXI4-Stream [29]. O protocolo AXI apresenta inúmeras funcionalidades e está disponível para qualquer utilizador que possua um processador ARM.

2.5.1 Serial Peripheral Interface

O SPI (*Serial Peripheral Interface*) é um protocolo que permite a comunicação entre um microcontrolador e diferentes componentes externos, sendo esta uma comunicação série síncrona com especificidade para distâncias curtas. A interface foi desenvolvida inicialmente pela Motorola e tornou-se cada mais comum devido à sua leve complexidade. Os dispositivos comunicam entre si usando uma arquitetura *master-slave*, sendo que tem apenas um único *master*. O *master* é que dita a comunicação e gera um sinal de *clock* denominado *Serial Clock* (SCLK). Para além deste sinal, existem duas linhas de dados, uma que faz a transmissão do *master* para o *slave* e outra do *slave* para o *master*, *Master Output Slave Input* (MOSI) e *Master Input Slave Output* (MISO), respetivamente. Ainda há um último sinal que funciona como *enable* que indica ao periférico quando deverá receber e enviar dados que se denomina de *Chip Select* (CS), também às vezes chamado de *Slave Select* (SS). Normalmente este sinal é *active low*, ou seja, o

sinal é mantido a '1' para não existir qualquer comunicação e quando o *master* quer comunicar, então transmite um sinal a '0' para iniciar a comunicação e mantêm-no durante esta mesma. Em situações em que existem mais do que um *slave* é necessário haver mais sinais SS, isto para o *master* especificar com que *slave* quer comunicar.

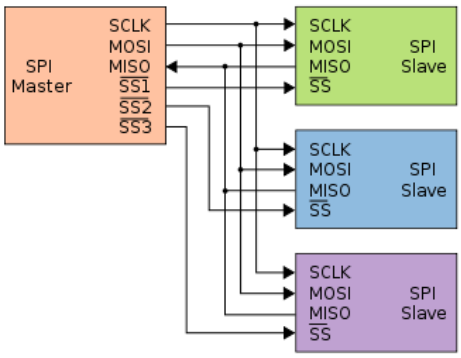

Figura 2.7: Modelação de ligações SPI num projeto com três *slaves* (*by* Cburnett, CC BY-SA 3.0).

A comunicação de dados através do protocolo SPI funciona da seguinte maneira:

1. O *master* gera e transmite o sinal de *clock* para o *slave*;
2. O *master* muda o sinal do SS para '0', o que faz com que o *slave* "acorde";
3. O *master* transmite um *bit* de cada vez para o *slave* através da linha MOSI e o *slave* vai lendo os *bits* enquanto os vai recebendo;
4. Caso seja necessária uma resposta, o *slave* retorna também um *bit* de cada vez pela linha MISO e o *master* também os lê à medida que vão chegando. Se for esse o caso o ponto 3 e 4 vão realizados em simultâneo.

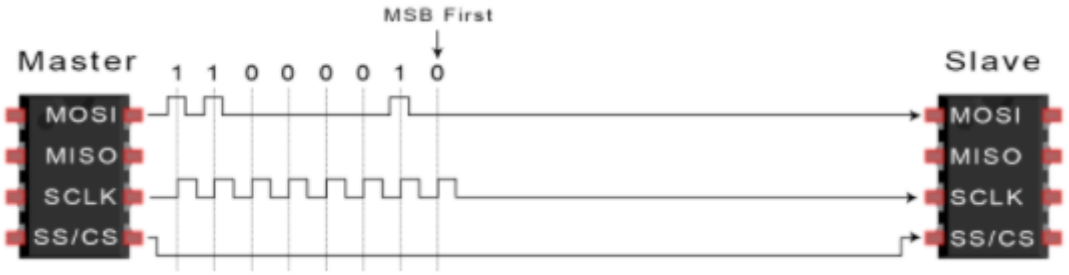

Figura 2.8: Ilustração de o *master* a enviar um *byte* para o *slave.* [30]

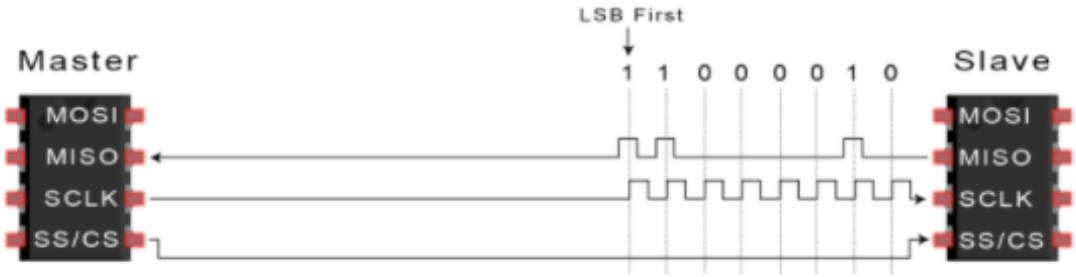

Figura 2.9: Ilustração de o *slave* a enviar um *byte* para o *master.* [30]

Normalmente o *master* ao mandar um *byte* para o *slave*, manda do MSB (*Most Significant Bit*) para o LSB (*Least Significant Bit*) e o *slave* obviamente irá recebê-los nesta ordem. No sentido inverso da comunicação, quando o *slave* envia um *byte* para o *master*, a situação também é ao contrário, pois o *slave* envia primeiro o LSB e por último envia o MSB.

O protocolo SPI foi utilizado em diversos projetos para suportar a comunicação entre FPGA e Raspberry Pi [31][32][33]. Por exemplo em [31] foi utilizado o protocolo SPI entre uma FPGA Spartan-3 XC3S200 e uma Raspberry Pi Model B para a medição de temperatura. Neste caso o Raspberry Pi é o *master* e a FPGA enquanto *slave*. Como o programa que controla o sistema foi escrito em linguagem C, para a implementação do SPI foram adicionadas bibliotecas, tais como, WiringPi e BMC2835. Nesta situação foi usada uma transmissão de 8 bits que na teoria daria para uma possibilidade máxima de 256 sensores de temperatura.

2.5.2 Inter-Integrated Circuit

O I2C (*Inter-Integrated Circuit*) é um barramento usado para conectar um *master* a múltiplos *slaves* ou vários *masters* a um ou mais *slaves*. Foi desenvolvido pela Phillips em 1982, com o intuito de criar um simples sistema de barramento interno nos seus chips eletrónicos. O I2C apena utiliza apenas duas linhas bidirecionais *Serial Data* (SDA) e *Serial Clock* (SCL).

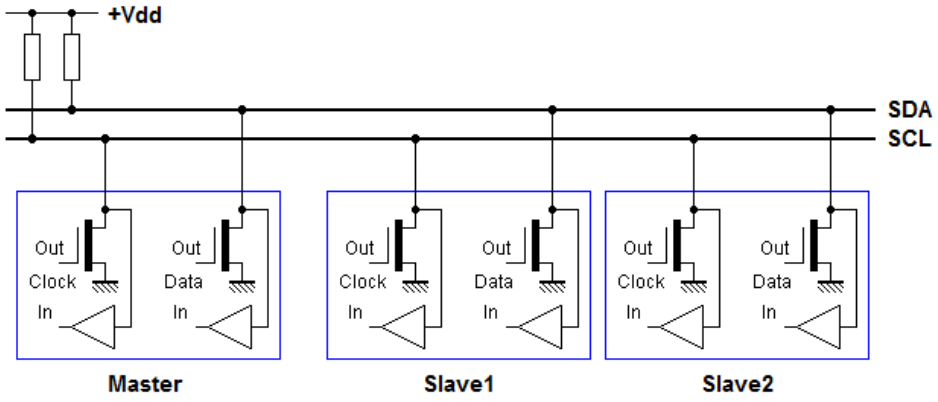

Figura 2.10: Ilustração das ligações entre *master* e *slaves* no protocolo I2C. [34]

O I2C é um protocolo de comunicação sequencial, portanto os dados são enviados *bit* a *bit* ao longo da linha de dados SDA. Tal como o SPI, o I2C é síncrono e, portanto, o sinal de *clock* SCL é gerado pelo *master* e é partilhado pelos *slaves*.

A mensagem transmitida pelo I2C é dividida em *frames* de dados. Cada mensagem tem um *frame* que funciona como endereço binário do *slave* com que quer iniciar comunicação e um ou mais *frames* de informação a transmitir. A mensagem também tem um *bit* de iniciar no começo da mensagem e um *bit* de stop no final. Para finalizar a mensagem ainda inclui um bit para transmite se o *master* quer ler ou escrever e *bits* de ACK/NACK entre cada *frame* de dados.

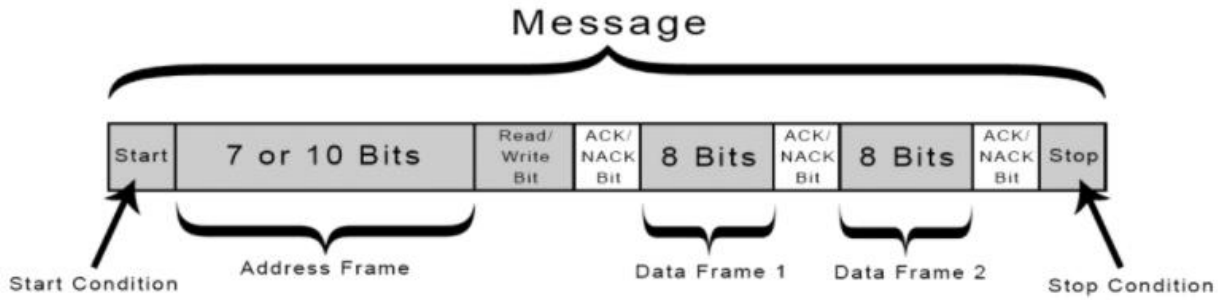

Figura 2.11: Exemplo de uma mensagem enviada por I2C. [35]

A comunicação de dados através do protocolo I2C funciona da seguinte forma, exemplificando passo a passo:

1. O *master* envia o *bit start* a todos os *slaves* ao qual está ligado;
2. De seguida envia o *frame* com o endereço do *slave* com que quer transmitir, juntamente com o bit de *read* ou *write*;
3. Cada *slave* compara o seu endereço com o endereço que o *master* acaba de transmitir. Se o endereço for igual, o *slave* envia um *bit* ACK enviando um '0' na linha de dados SDA. Caso o endereço não coincida, o *slave* continua com a o sinal SDA *active high*;
4. O *master* envia ou recebe o *frame* de dados pela linha SDA;
5. A cada *frame* de dados, o recetor envia um *bit* de ACK (*active low*) para que se saiba que a receção está a ser feita com sucesso;
6. Para finalizar a transmissão, o *master* envia o *bit* de stop.

Um exemplo de um projeto em que se usou o protocolo I2C, para conectar uma FPGA Spartan 3 a um circuito integrado DS1307 que funciona como um *real-time clock* que suporta duas linhas bidirecionais e este protocolo de transmissão, foi o apresentado em [36]. A FPGA funciona como *master* e o DS1307 como *slave*.

Como podemos observar os dois protocolos descritos acima têm algumas divergências no que toca ao número de linhas necessárias para a realização do protocolo. Ambos são síncronos, mas o SPI transfere dados mais rapidamente do que o I2C e não tem a complexidade dos endereços de *slaves* e os *bits* de start e stop, nem os *bits* ACK e, portanto, os dados podem ser enviados de uma forma mais contínua sem interrupções. Como tem duas linhas de dados separadas, o SPI consegue enviar e receber dados simultaneamente. Mas por outro lado, o I2C consegue ter múltiplos masters caso necessário e tem a verificação se os dados estão a ser recebidos corretamente.

3

3. Sistema Proposto

Este capítulo descreve o sistema proposto. Este foi pensado e implementado de modo a processar imagens em tempo real de uma maneira acessível e simples para quem queira realizar este tipo de processamento.

Na secção 3.1 faz-se uma breve introdução sobre as características deste sistema e protocolo de comunicação utilizado. Na secção 3.2 apresenta-se o *hardware* utilizado: câmara, FPGA e Raspberry Pi. Nas secções 3.3 e 3.4 descreve-se as ligações entre a câmara e a FPGA e entre a FPGA e o Raspberry Pi, respetivamente. O módulo SPI master e o módulo SPI *slave* são apresentados nas secções 3.5 e 3.6 respetivamente. A secção 3.7 descreve o funcionamento do sistema e por fim a secção 3.8 apresenta o esquemático do projeto no Vivado.

3.1 Introdução

Após concebido o objetivo e após a pesquisa feita durante o estado de arte, ficou claro que o projeto se ira focar em três componentes: a FPGA, o computador e a comunicação entre eles. Posto isto, sabendo que a FPGA tem como vantagem a velocidade de execução e capacidade de executar tarefas em paralelo, mas que tem elevada complexidade ao nível de implementação de algoritmos, a FPGA irá realizar a aquisição das imagens, pré-processamento e envio dados para o computador. O restante processamento de imagem será realizado pelo computador. Na figura 3.1 é apresentado o diagrama de blocos do sistema proposto. Como mostra a figura, o sistema é composto pela câmara que transmite a imagem adquirida à FPGA e que por sua vez vai transmitir a imagem já pré-processada e algumas características para o computador.

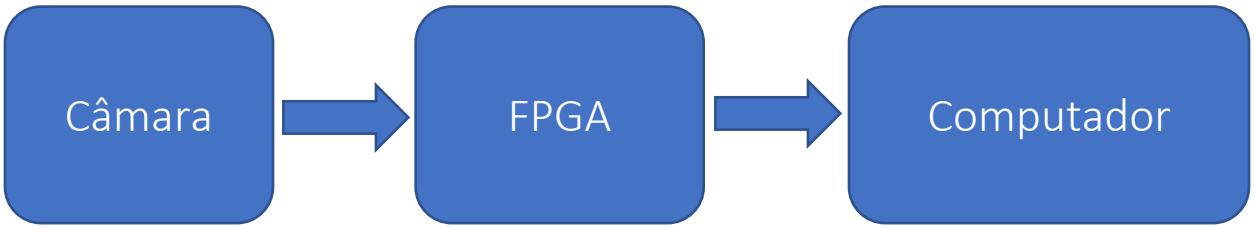

Figura 3.1: Diagrama de blocos de uma plataforma heterogénea para aquisição e processamento de imagem.

A plataforma que se propõe combina uma FPGA Basys 3 e um Raspberry Pi 3, isto devido ao facto de o Raspberry Pi ser um computador bastante acessível com uma enorme fonte de informação disponível, com uma forte comunidade que partilha os seus projetos, as suas ideias, os seus tutoriais e as suas dificuldades. A câmara estará conectada à FPGA, que tem o objetivo que fazer um pré-processamento da imagem adquirida e que posteriormente será transmitida para o Raspberry Pi através de SPI (*Serial Peripheral Interface*). O intuito final desta plataforma é disponibilizar o projeto versátil para que ajude e seja mais fácil a outras pessoas utilizarem uma FPGA e um Raspberry Pi para realizar processamento de imagem. No Raspberry Pi 3 o

processamento poderá ser feito através da biblioteca OpenCV em Python, usando por exemplo o IDE Thonny.

3.2 Hardware Utilizado

A câmara utilizada (figura 3.2) para a captura das imagens deste sistema é o módulo OV7670 [37]. Uma pequena câmara CMOS que trabalha a uma pequena diferença de potencial. Esta câmara tem uma taxa de atualização de até 30 *frames* por segundo para VGA, a uma resolução máxima de 640 x 480 pixéis. O seu output pode ser em formato YUV, YCbCr 4:2:2, RGB565, RGB555, ou GRB 4:2:2. O formato do output usado neste trabalho foi o RGB565.

Figura 3.2: Imagem da câmara OV7670.

A FPGA usada neste projeto foi uma Basys 3 Artix-7 FPGA [38] (figura 3.3), que é exclusivamente desenhada para o Vivado Design Suite. Como o nome indica tem uma Xilinx Artix-7, tem também um elevado número de componentes de entrada e saída: 16 LED, 16 interruptores e 5 botões de pressão. A FPGA tem uma saída VGA e 4 conectores Pmod e 1,800 Kbits de Block RAM. Estes conectores Pmod vão ser necessários para realizar a ligação entre a FPGA e a câmara e ligar a FPGA ao computador.

Figura 3.3: Imagem da Basys 3 Artix-7 FPGA.

Como referido anteriormente a FPGA Basys 3 está associada ao Vivado Design Suite, portanto vai ser usado este *software* produzido pela Xillinx para síntese e análise de modelos HDL na plataforma da dissertação. A versão usada, apresentada na figura 3.4, foi a 2019.1.

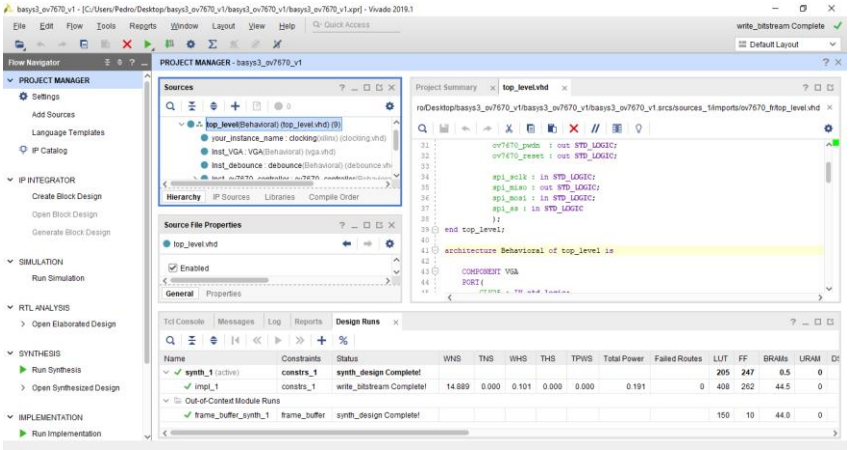

Figura 3.4: Imagem do Vivado Design Suite 2019.1.

Em relação ao computador, foi usado um Raspberry Pi 3. Este computador de tamanho reduzido, tem suporte para o protocolo SPI e I2C, tem 40 pins GPIO, para além de fichas USB onde se pode ligar teclado e rato, HDMI (para ligar um monitor), entre outras, como apresentado na figura 3.5.

Figura 3.5: Ilustração do modelo Raspberry Pi 3.

3.3 Ligação entre a Câmara e a FPGA

Esta parte do trabalho foi de certa forma mais simples, pois encontrou-se um projeto [39] que realizava esta interface. Posteriormente este projeto foi adaptado em concordância com os objetivos deste trabalho. As ligações entre a câmara e a FPGA são idênticas ao projeto referido.

A Basys 3 (figura 3.6) tem quatro Pmods: JA, JB, JC e JXADC. Este último não é usado neste sistema pois está relacionado com *inputs* para realizar a conversão entre analógico e digital. Cada Pmod tem 12 pins em que cada um é composto por 8 pins de sinais e 2 pins de 3,3V VCC e 2 pins de *ground*.

Figura 3.6: Imagem do datasheet da Basys 3 mostrando os seus pins Pmod. (adaptado de [38] e [39])

A câmara tem 18 pins de comunicação sendo que oito deles são relativos a transmissão de dados e é alimentada pela FPGA. Os pins de entrada da FPGA são fêmea e os pins da câmara são macho, portanto teve de se usar fios macho-fêmea deste procedimento.

As ligações entre a FPGA e a câmara foi realizada usando os conectores Pmod JB e JC, como mostra a figura 3.7. Em relação ao pin 3v3 e ao GND da câmara que não foram ilustrados na figura, estes dois ligaram-se diretamente ao JC12 e JC11 respetivamente.

Figura 3.7: Diagrama de ligações entre a câmara e a FPGA usando os conectores Pmod. [39]

A figura 3.8 mostra as ligações realizadas entre a câmara e a FPGA do protótipo concebido para a presente dissertação. Como podemos observar a câmara não tem muito espaço de manobra para se mover, mas para este efeito chega para realizar os testes e validar o proposto.

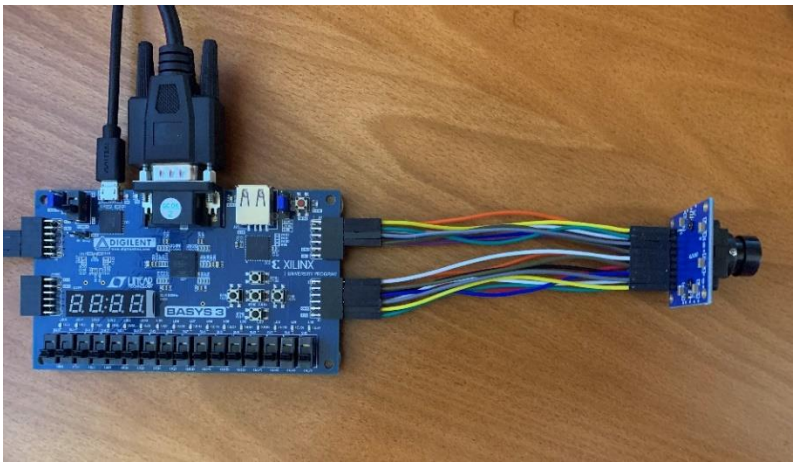

Figura 3.8: Imagem das ligações entre a câmara e a FPGA do sistema.

A figura 3.9 apresenta-se código do ficheiro .xdc que inclui que pinos vão ser usados e qual a sua relação com o código VHDL descrito para realizar o processamento dos pixéis. Os pins de alimentação da câmara, 3V3 e o GND não são descritos nesse ficheiro, pois apenas servem para manter a câmara ativa, não existe qualquer manipulação desse sinal.

```
##Pmod Header JB
##Sch name = JB1
set_property PACKAGE_PIN A14 [get_ports {ov7670_pwdn}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_pwdn}]
##Sch name = JB2
set_property PACKAGE_PIN A16 [get_ports {ov7670_data[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[0]}]
##Sch name = JB3
set_property PACKAGE_PIN B15 [get_ports {ov7670_data[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[2]}]
##Sch name = JB4
set_property PACKAGE_PIN B16 [get_ports {ov7670_data[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[4]}]
##Sch name = JB7
set_property PACKAGE_PIN A15 [get_ports {ov7670_reset}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_reset}]
##Sch name = JB8
set_property PACKAGE_PIN A17 [get_ports {ov7670_data[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[1]}]
##Sch name = JB9
set_property PACKAGE_PIN C15 [get_ports {ov7670_data[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[3]}]
##Sch name = JB10
set_property PACKAGE_PIN C16 [get_ports {ov7670_data[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[5]}]

```

```
##Pmod Header JC
##Sch name = JC1
set_property PACKAGE_PIN K17 [get_ports {ov7670_data[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[6]}]
##Sch name = JC2
set_property PACKAGE_PIN M18 [get_ports ov7670_xclk]
    set_property IOSTANDARD LVCMOS33 [get_ports ov7670_xclk]
##Sch name = JC3
set_property PACKAGE_PIN N17 [get_ports ov7670_href]
    set_property IOSTANDARD LVCMOS33 [get_ports ov7670_href]
##Sch name = JC4
set_property PACKAGE_PIN P18 [get_ports ov7670_siod]
    set_property IOSTANDARD LVCMOS33 [get_ports ov7670_siod]
    set_property PULLUP TRUE [get_ports ov7670_siod]
##Sch name = JC7
set_property PACKAGE_PIN L17 [get_ports {ov7670_data[7]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ov7670_data[7]}]
##Sch name = JC8
set_property PACKAGE_PIN M19 [get_ports ov7670_pclk]
    set_property IOSTANDARD LVCMOS33 [get_ports ov7670_pclk]
    set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets {ov7670_pclk_IBUF}]
##Sch name = JC9
set_property PACKAGE_PIN P17 [get_ports ov7670_vsync]
    set_property IOSTANDARD LVCMOS33 [get_ports ov7670_vsync]
##Sch name = JC10
set_property PACKAGE_PIN R18 [get_ports ov7670_sioc]
    set_property IOSTANDARD LVCMOS33 [get_ports ov7670_sioc]
```

Figura 3.9: Parte de código para mapear os pins da FPGA.

3.4 Ligação entre a FPGA e o Raspberry Pi

O Raspberry Pi 3 tem 2 filas de 20 pins, portanto um total de 40 pins, sendo que apenas 26 deles são usados como GPIO (figura 3.10). Os restantes pins são de alimentação 3v3, 5V, *Ground.* A vantagem do Raspberry Pi é que já tem pins definidos para o protocolo SPI e I2C, o que ajuda bastante a implementação.

O SPI tem dois canais específicos no Raspberry Pi:

- SPI 0: MOSI (GPIO 10); MISO (GPIO 9); SCLK (GPIO 11); CE0 (GPIO 8); CE1 (GPIO 7)
- SPI 1: MOSI (GPIO 20); MISO (GPIO 19); SCLK (GPIO 21); CE0 (GPIO 18); CE1 (GPIO 17); CE2 (GPIO 16)

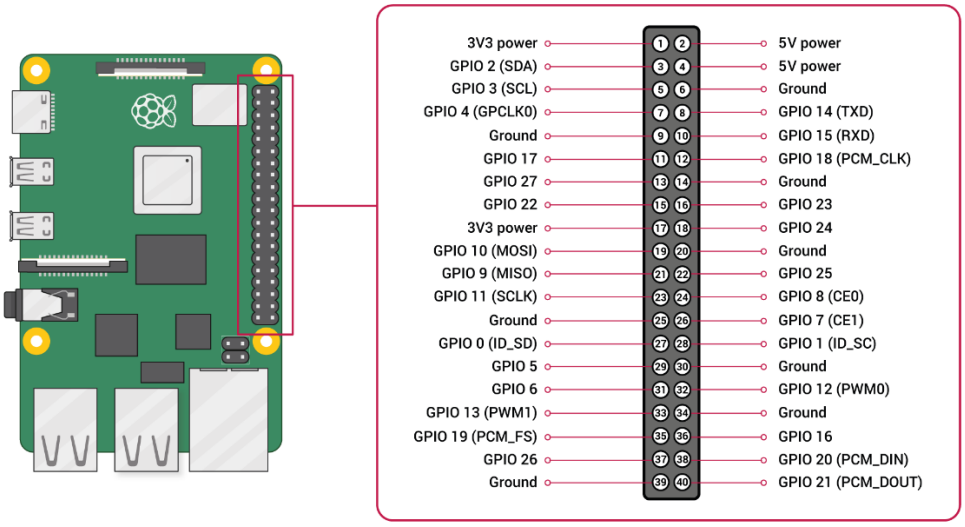

Figura 3.10: Imagem do mapeamento dos pins GPIO do Raspberry PI 3. [40]

Neste sistema seria indiferente qual o canal SPI a usar, pois apenas se tem um *master* a comunicar com um *slave*. Foi usado o SPI bus 0 e, portanto, criou-se ligações aos pins acima descritos (o CE0 foi usado como *Slave Select*). Na figura 3.11 apresenta-se parte do código Python que inicializa o canal SPI 0 (*bus*) com o CE0 (*Slave Select*) no Raspberry PI. Como se pode ver na figura, foi utilizada a biblioteca Python Spidev [41] para o Raspberry PI comunicar por SPI.

```
1  import spidev
2  import time
3
4  #We only have SPI bus 0 available to us on the Pi
5  bus = 0
6
7  #Device is the chip select pin, Set to 0 or 1, depending on the connections
8  device = 0
9
10 #Enable SPI
11 spi = spidev.SpiDev()
12
13 #Open a connection to a specific bus and device (chip select pin)
14 spi.open(bus, device)
```

Figura 3.11: Parte do código que inicializa o canal SPI 0 com o CE0.

Do lado da FPGA, os quatro sinais foram incluídos no Pmod JA e mais uma vez foram descritos no ficheiro .xdc do projeto para que possa manipular as entradas e saídas provenientes do Raspberry Pi. Os quatro sinais SCLK, MISO, MOSI e SS estão conectados aos pins JA1, JA2, JA3 e JA4 respetivamente. Além desses quatro sinais, ligou-se também um GND entre o Raspberry Pi e a FPGA. Na figura 3.12 apresenta-se parte de código para mapear os pins associados ao SPI na FPGA.

```
##Pmod Header JA
##Sch name = JA1
set_property PACKAGE_PIN J1 [get_ports {spi_sclk}]
    set_property IOSTANDARD LVCMOS33 [get_ports {spi_sclk}]
##Sch name = JA2
set_property PACKAGE_PIN L2 [get_ports spi_miso]
    set_property IOSTANDARD LVCMOS33 [get_ports spi_miso]
##Sch name = JA3
set_property PACKAGE_PIN J2 [get_ports spi_mosi]
    set_property IOSTANDARD LVCMOS33 [get_ports spi_mosi]
##Sch name = JA4
set_property PACKAGE_PIN G2 [get_ports spi_ss]
    set_property IOSTANDARD LVCMOS33 [get_ports spi_ss]
```

Figura 3.12: Parte de código para mapear os pins associados ao SPI na FPGA.

A figura 3.13 mostra a ligação SPI do sistema apresentado, como já referido anteriormente os fios para suportar o protocolo proposto.

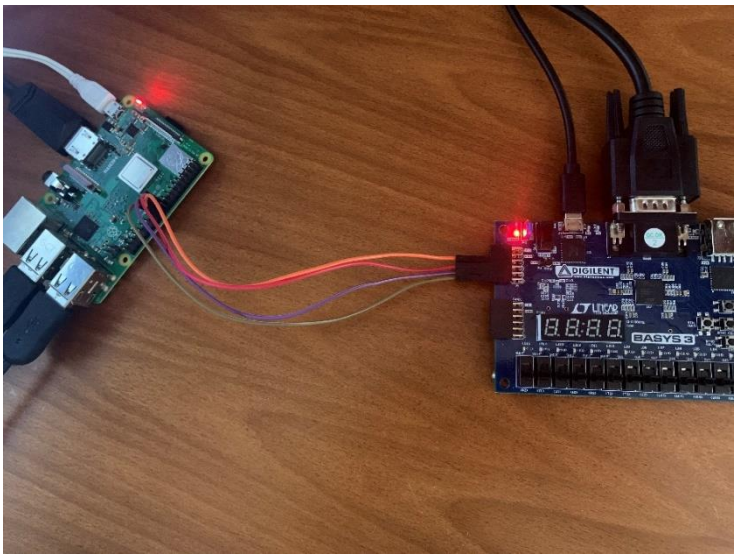

Figura 3.13: Imagem das ligações entre a FPGA e o Raspberry Pi.

3.5 Módulo SPI Master

Para testar o sistema desenvolvido, foi criado um módulo SPI *master* no Raspberry Pi. O módulo do SPI *master* de certa forma é mais fácil de implementar uma vez que o Raspberry Pi tem o seu próprio módulo de interface com dispositivos usando o protocolo SPI. Este módulo tem bastantes métodos disponibilizados e existe bastante documentação em como implementar num ambiente de desenvolvimento Python. Um dos métodos que foram usados neste projeto é o *spi.xfer2*, da biblioteca spidev [41], que realiza uma transação de informação SPI entre o *master* e o *slave*, com a particularidade de o sinal *chip-select* (*slave select*) ser desativado e reativado entre blocos de transmissão.

Na figura 3.14, que apresenta um exemplo de código utilizado, o código 0xc7 é enviado para que a FPGA saiba que o protocolo foi inicializado e posteriormente realiza um ciclo reconstruindo a imagem, que neste caso é uma imagem 320x240 em tons de cinzento. O código 0x07 é enviado ciclicamente para pedir cada um dos pixéis, iterando os todos pixéis de uma linha, que por sua vez itera todas as linhas da imagem.

```
21  #Set SPI speed and mode
22  spi.max_speed_hz = 4000000
23  spi.mode = 0
24
25  w, h = 320, 240
26  data = np.zeros((h, w, 3), dtype=np.uint8)
27
28  msgInitial = [0xc7]
29  [resultInitial] = spi.xfer2(msgInitial)
30  time.sleep(0.001)
31
32  for y in range(h):
33
34      for x in range(w):
35
36          msg = [0x07]
37          [result] = spi.xfer2(msg)
38
39          data[y, x] = [result, result, result]
40
41  img = Image.fromarray(data, 'RGB')
42  img.save('Teste.png')
43  img.show()
```

Figura 3.14: Código implementado no Raspberry Pi para recriar imagem vinda do *slave*.

3.6 Módulo SPI *Slave*

Nesta secção é apresentado o código VHDL utilizado para implementar o protocolo SPI. Tal como foi referido anteriormente a FPGA tem um módulo SPI *slave* que recebe e transmite informação ao *master*. No capítulo 2 foi apresentada a forma como este protocolo atua e os sinais necessários para a sua implementação.

No momento em que o *master* quer enviar um *byte*, coloca o sinal *slave select* a '0' e a partir desse momento, regendo-se pelo *clock*, que também gera, vai alterando o sinal MOSI de acordo com o *byte* que quer enviar. O *slave*, que está constantemente a analisar o sinal *slave select* e o sinal de *clock*, apercebe-se que a comunicação iniciou quando o sinal *slave select* passar de '1' para '0' e a partir desse momento sempre que o sinal de clock passa de '1' para '0', como se pode ver na linha 55 da figura 3.15, um bit é guardado num registo de deslocamento. A contagem do número de bits recebidos é feita da linha 50 à linha 54 da figura 3.15. Ao receber o oitavo bit, o *slave* sabe que recebeu o *byte* completo e nesse momento irá colocar o *byte* recebido na saída do módulo e colocar um sinal a '1' para indicar que um byte foi recebido. Esse valor será passado para o módulo que faz o processamento dos comandos e valores recebidos, descrito na seção 3.7.

```
38  rx_state_machine: process(CLK)
39  begin
40    if rising_edge(CLK) then
41      if s_rst = '1' then
42         rx_state <= 0;
43          rx_byte <= "00000000";
44       elsif s_ssn = '1' then
45         rx_state <= 1;
46          rx_count <= 0;
47       elsif rx_state = 1 and s_ssn = '0' then
48         rx_state <= 2;
49       elsif rx_state = 2 and prev_sclk = '0' and s_sclk = '1' then
50         if rx_count < 7 then
51           rx_count <= rx_count + 1;
52         else
53           rx_count <= 0;
54         end if;
55          rx_byte <= rx_byte(6 downto 0) & s_mosi;
56       end if;
57    end if;
58  end process;
```

Figura 3.15: Código VHDL que controla a receção dos bits.

A lógica do processo de transmissão é semelhante à lógica do processo de receção. Para além de se verificar a passagem do sinal de *slave select* de '1' para '0', verifica-se também a passagem do sinal de *clock* de '1' para '0', como se pode ver na linha 104 da figura 3.16, e faz-se a contagem dos bits enviados e coloca-se o bit a enviar no sinal respetivo (linha 110).

```
91  tx_state_machine: process(CLK)
92  begin
93    if rising_edge(CLK) then
94      if s_rst = '1' then
95         tx_state <= 0;
96          miso_reg <= '0';
97       elsif s_ssn = '1' then
98         tx_state <= 1;
99          miso_reg <= '0';
100      elsif tx_state = 1 and s_ssn = '0' then
101        tx_state <= 2;
102         miso_reg <= tx_byte(7);
103         tx_count <= 1;
104      elsif tx_state = 2 and prev_sclk = '1' and s_sclk = '0' then
105        if tx_count < 7 then
106            tx_count <= tx_count + 1;
107        else
108            tx_count <= 0;
109        end if;
110        miso_reg <= tx_byte(7-tx_count);
111     end if;
112   end if;
113 end process;
```

Figura 3.16: Código VHDL que guarda os bits recebidos e gera os bits a enviar.

3.7 Funcionamento do Sistema

Relativamente à interação entre a FPGA e o Raspberry Pi via SPI, o SPI *master* vai ser o Raspberry Pi e na FPGA vai existir um módulo que vai fazer essa comunicação como *slave*. Enquanto a FPGA vai adquirindo a imagem capturada pela câmara, vai recebendo instruções dadas pelo Raspberry Pi. Instruções essas que dependendo do código enviado pelo *master*, o *slave* vai enviar o próximo pixel com o pré-processamento solicitado.

Cada tipo de processamento na FPGA está associado a um código, logo durante o processo de transmissão por SPI, o master inicia a comunicação enviando um código associado ao processamento que pretende. Após o *slave* receber o byte com o código, devolve na comunicação um byte com o próximo pixel processado. Na figura 3.17 apresenta-se o Diagrama de sequência da interação entre o computador e a FPGA.

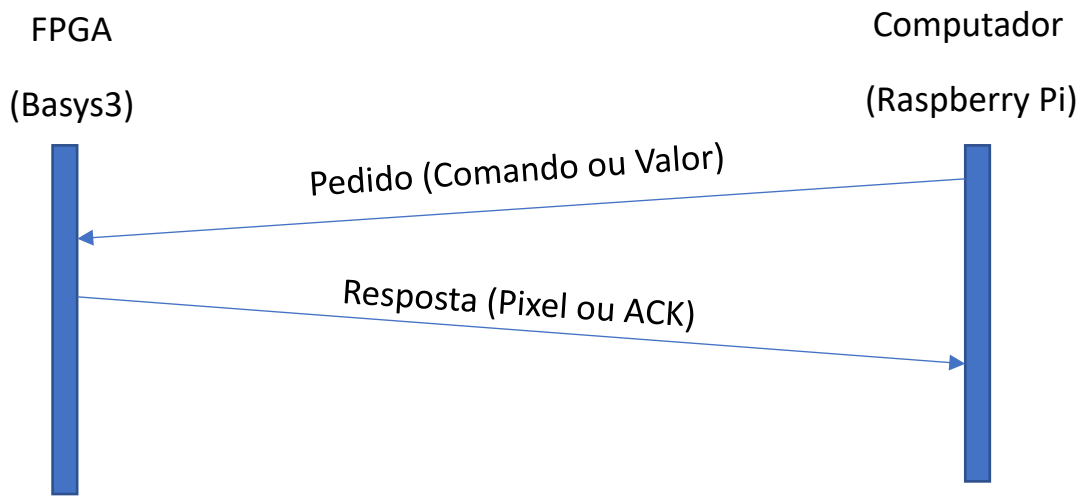

Figura 3.17: Diagrama de sequência da interação entre o computador (*master*) e a FPGA (*slave*).

O pedido (ver figura 3.17) enviado pelo computador pode conter um comando ou um valor. Quando é comando, o *byte* está dividido em três partes. Nos 3 bits mais significativos está o comando (ver tabela 3.1). Nos 3 *bits* menos significativos do comando indicam, relativamente ao pixel que se pretende receber no próximo pedido, as componentes de cor ou se se pretende converter o pixel para tons de cinzento e fazer processamento adicional (ver tabela 3.2). O *bit* 4 do comando indica se se pretende converter o pixel (em tons de cinzento) para negativo, enquanto que o bit 3 do comando indica se se pretende binarizar o pixel (que está em tons de cinzento). Quando se trata de um comando de configuração, no pedido seguinte o computador envia o valor correspondente, que será o *threshold* de binarização, o valor de aumento de brilho ou o valor de diminuição de brilho. Na figura 3.18 apresenta-se o diagrama de sequência de configuração de um valor, seguido de pedido de pixel.

Tabela 3.1: Três bits mais significativos do comando.

Bits 7,6 e 5 do comando	Descrição
000	Pedido de pixel atual
001	Configuração do *threshold* de binarização
010	Configuração de aumento de brilho
011	Configuração de diminuição de brilho
100	Retirar contraste
101	Aumentar o contraste (multiplica por 2)
110	Diminuir o contraste (divide por 2)
111	Coloca o apontador da imagem no início (0,0)

Tabela 3.2: Três bits menos significativos do comando.

Bits 2,1 e 0 do comando	Descrição
000	*Red channel*
001	*Green channel*
010	*Blue channel*
011	*Red and Green channels*
100	*Red and Blue channels*
101	*Green and Blue channels*
110	*Red, Green, and Blue channels*
111	Conversão para tons de cinzento

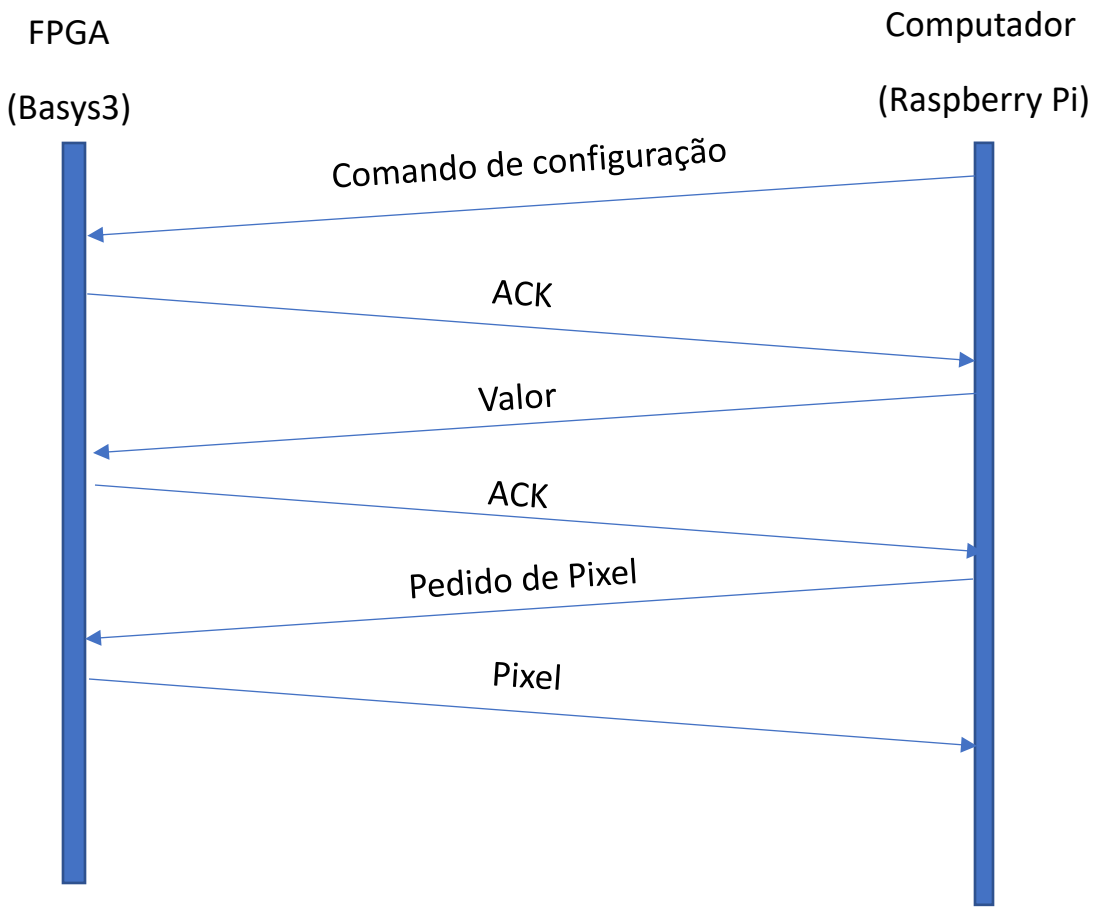

Figura 3.18: Diagrama de sequência de configuração e pedido de pixel.

Na tabela 3.3 apresenta-se uma lista de códigos, que a FPGA será capaz de receber e interpretar. Os códigos são apresentados em código hexadecimal.

Tabela 3.3: Lista de códigos que a FPGA será capaz de receber e interpretar, com a respetiva descrição.

Código	Descrição	Código	Descrição
0x00	Pedido de Red Channel	0x0F	Pedido de pixel em tons de cinzento binarizado
0x01	Pedido de Green Channel	0x1F	Pedido de pixel em tons de cinzento binarizado e negativo
0x02	Pedido de Blue Channel	0x17	Pedido de pixel em tons de cinzento negativo
0x03	Pedido de Red e Green Channels	0xA7	Pedido de pixel em tons de cinzento com contraste duplicado por 2
0x04	Pedido de Red e Blue Channels	0xC7	Pedido de pixel em tons de cinzento com contraste dividido por 2
0x05	Pedido de Green e Blue Channels	0x27	Configuração do *threshold* de binarização (no próximo pedido é enviado o *threshold*)
0x06	Pedido de RGB	0x47	Configuração de aumento de brilho (no próximo pedido é enviado o brilho)
0x07	Pedido de pixel em tons de cinzento (média ponderada)	0x67	Configuração de diminuição de brilho (no próximo pedido é enviado o brilho)

A figura 3.19 apresenta o diagrama de blocos relativo ao processamento do pixel pela FPGA. Do lado esquerdo da imagem entra o pixel com 12 bits, 4 para cada componente de cor. Dependendo do valor dos 3 *bits* menos significativos do comando, será colocado à saída uma ou várias componentes de cor, ou o pixel será processado. Por exemplo, quando se pretende enviar apenas a componente vermelha, e como é sempre enviado um valor com 8 *bits*, vai-se concatenar a componente com 4 zeros à direita. Quando o valor é convertido para cinzento (3 *bits* menos significativos do comando a um) pode-se adicionalmente multiplicar por 2, dividir por 2, aumentar o brilho, diminuir o brilho, converter para preto ou branco e fazer o negativo.

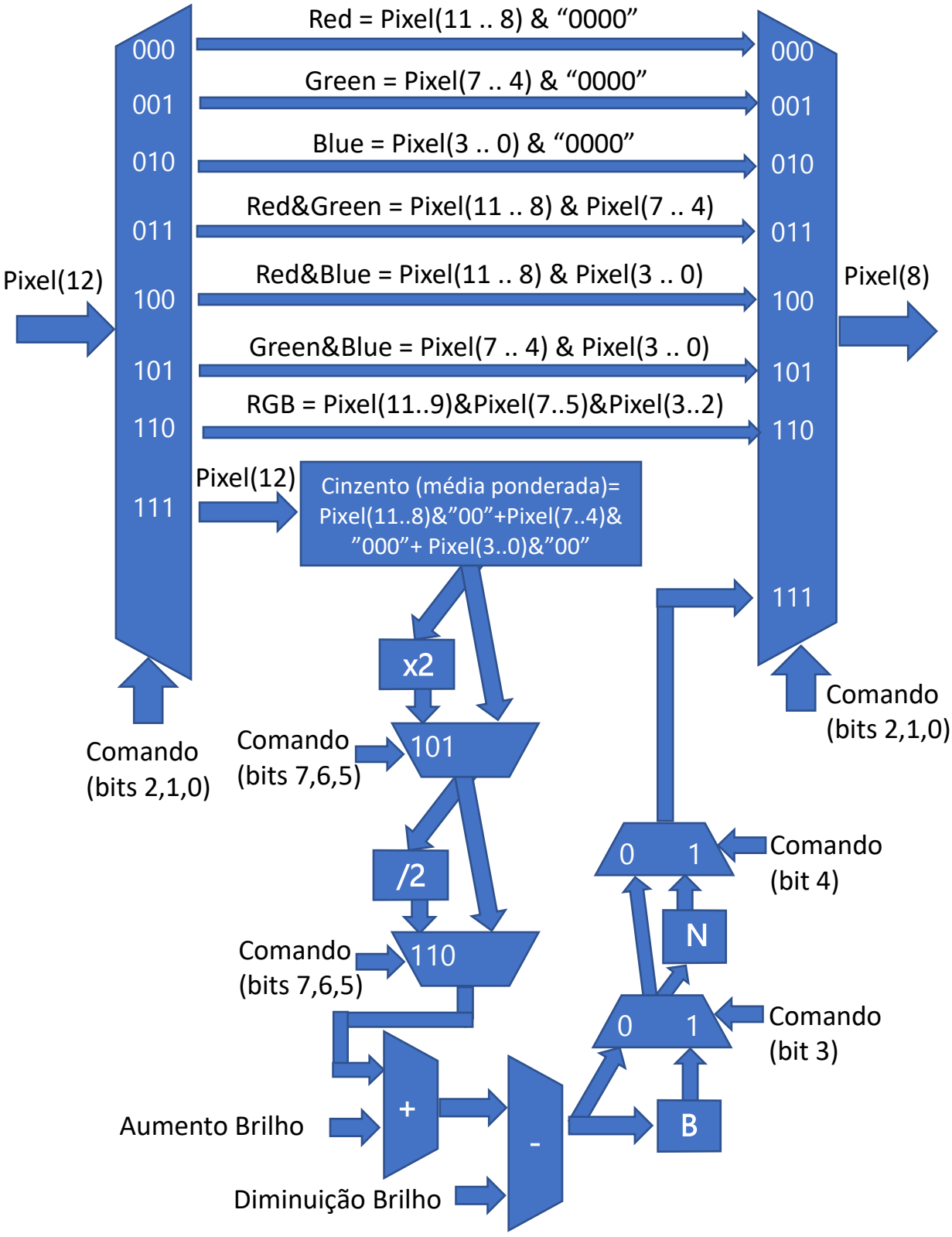

Figura 3.19: Diagrama de blocos do processamento do pixel.

O diagrama da figura 3.20 mostra a ligação do componente de processamento com o componente SPI *slave* e a ligação da FPGA com o Raspberry Pi (SPI *master*). representa o mesmo diagrama, mas agora numa vertente de maior nível em que podemos observar que o Raspberry Pi, enquanto *master*, não tem qualquer contacto com as diferentes funções acima indicadas neste capítulo. O *master* apenas "olha" para o envio e receção de dados na comunicação com o módulo *slave* inserido na FPGA.

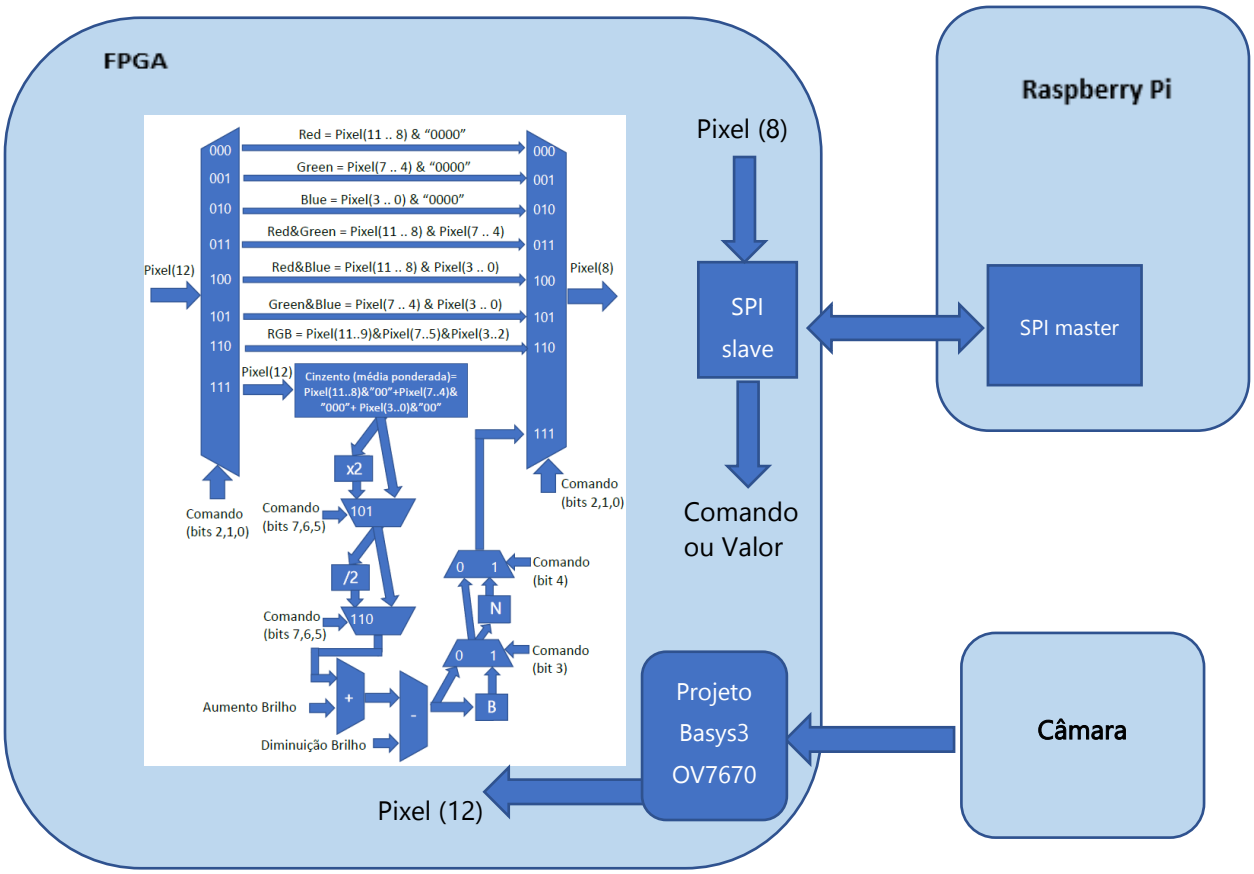

Figura 3.20: Diagrama de blocos de topo.

Desta forma, o Raspberry Pi poderá configurar a FPGA para que está faça o pré-processamento desejado. Ao receber os pixéis da imagem o Raspberry Pi poderá então efetuar o restante processamento, por exemplo em Python e com recurso à biblioteca de processamento de imagem OpenCV.

3.8 Esquemático do Sistema na FPGA

Nesta secção é apresentado o esquemático final do projeto, que mostra os vários módulos. O bloco *clocking* gera dois *clocks* com 50MHz e 25MHz a partir do clock da FPGA de 100 MHz, isto porque a frequência da interface de captura de imagem da câmara é de 50 MHz e a frequência que a FPGA utiliza para mostrar a imagem no monitor através de VGA é de 25 MHz. O bloco relativo à VGA tem o intuito de fazer a sincronização da VGA de modo a transmitir a imagem para o monitor. O bloco de SPI *slave* vai realizar a comunicação entre a FPGA e o Raspberry Pi. Este tanto recebe um *byte* vindo do *master* como envia um *byte* para o *master*, mantendo a transmissão bidirecional (tanto recebe códigos provenientes do Raspberry Pi, como posteriormente envia o pixel suposto). Tanto o bloco VGA como o bloco SPI estão conectados ao bloco RGB que processa o pixel consoante os códigos recebidos do Raspberry Pi (ver seção 3.7). Os outros blocos que estão presentes na figura 3.21 são blocos relativos à captura de imagem por parte da câmara OV7670 e a sua interface com a FPGA, estes blocos já estavam inseridos no projeto disponível em [39].

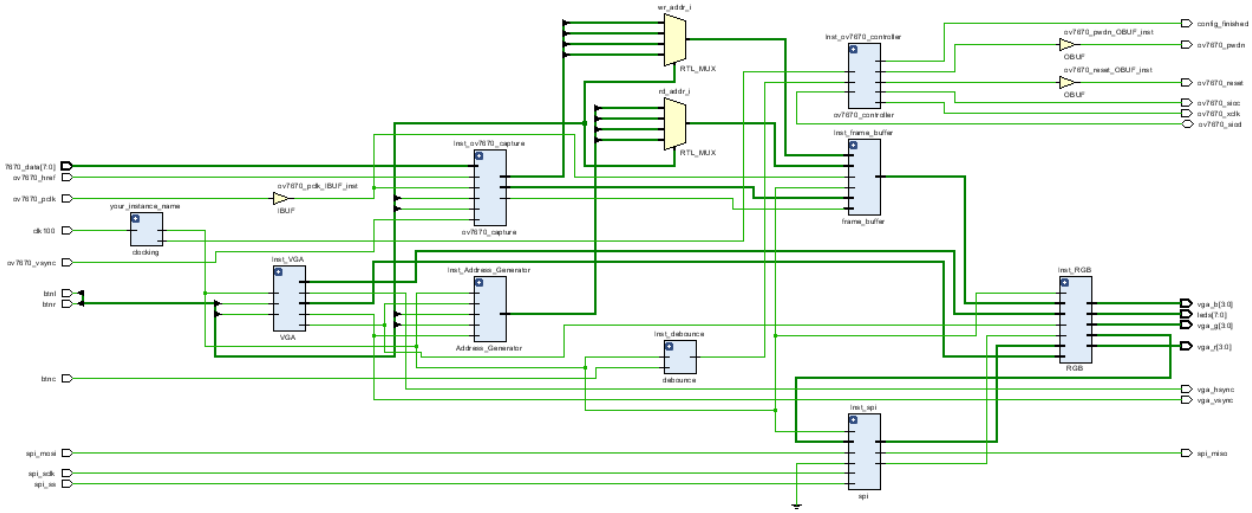

Figura 3.21: Esquemático do projeto no Vivado.

4

4. Testes e Resultados

Este capítulo apresenta os testes efetuados e os resultados obtidos. Na secção 4.1 apresenta-se uma introdução ao capítulo. Os testes realizados e respetivos resultados são apresentados na secção 4.2, juntamente com partes de código necessário para implementar as funções descritas no capítulo 3. Por fim na secção 4.3 apresentam-se os recursos utilizados na FPGA.

4.1 Ambiente de Teste

Para além da câmara, FPGA e Raspberry Pi, para realizar os testes foram ainda utilizados um monitor, um teclado e um rato, como ilustrado na figura 4.1. A FPGA e o Raspberry Pi foram ligados ao mesmo monitor, sendo possível visualizar a imagem saindo da FPGA depois de um pré-processamento e o resultado após o processamento final no Raspberry Pi. O teclado e o rato permitiram interagir com o Raspberry. As imagens recebidas no Raspberry Pi foram também guardadas em disco e são apresentadas ao longo deste capítulo.

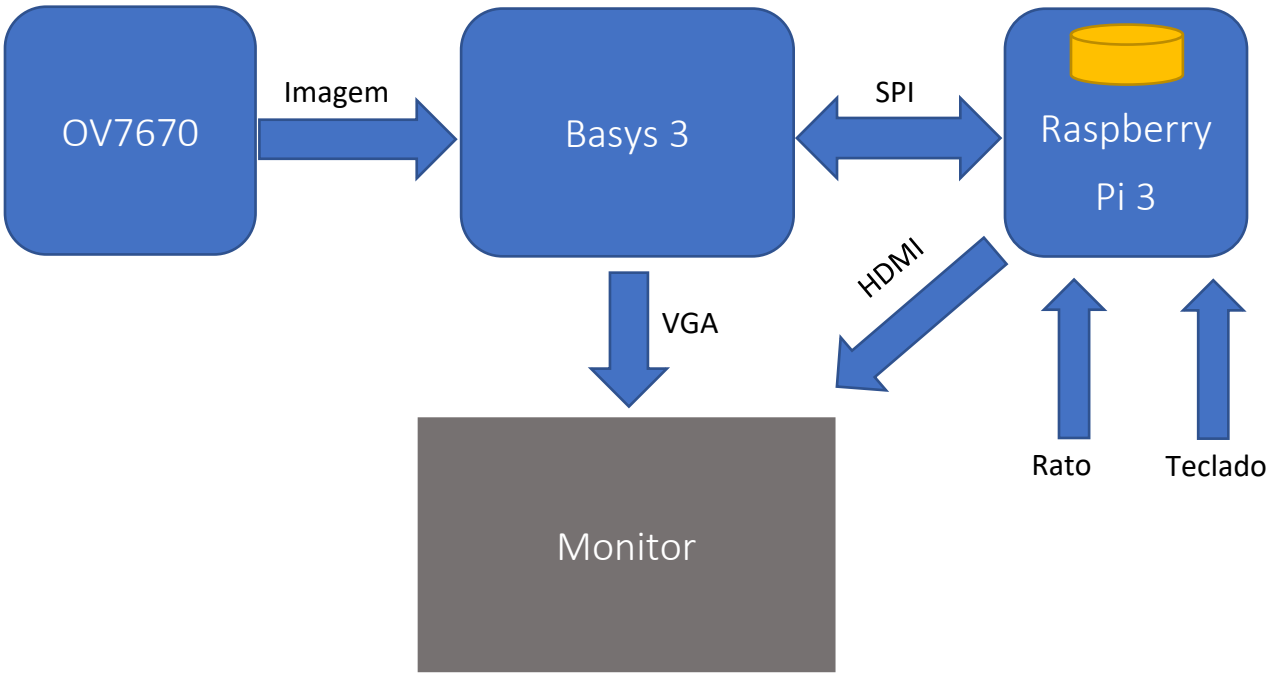

Figura 4.1: Ambiente de teste.

4.2 Testes

Nesta secção do capítulo apresentam-se os testes que foram realizados em termos de processamento dos pixéis e manipulação a imagem adquirida pela câmara. Além disso foram feitos testes ao protocolo SPI e verificou-se que o módulo *slave* e o módulo *master* funcionavam perfeitamente até a uma frequência de 4Mhz. Passando esta frequência, começava a haver incoerências na transmissão e divergências quanto ao *byte* transmitido e ao suposto *byte* recebido.

4.2.1 Testes de frequência na comunicação da FPGA para o Raspberry

De modo a compreender qual deveria ser a frequência máxima a ser utilizada pelo Raspberry Pi mantendo a eficácia de transmissão foi realizado o seguinte teste. O valor binário dos primeiros oito interruptores da FPGA é enviado para o Raspberry Pi usando o protocolo SPI. Neste teste vamos fazer variar o número de mensagens e a frequência de comunicação e registar o número de erros que ocorreram em cada teste. Este foi feito **sem *delay*** entre o envio sucessivo de mensagens. O *byte* enviado foi "01001101" (77 em decimal), escolhido aleatoriamente.

Como podemos observar pela tabela 4.1, até uma frequência de 4 MHz o comportamento da transmissão de informação é viável, não tendo qualquer falha na comunição. Para uma frequência de 5 MHz notou-se falhas ao nível do *byte* recebido e mostrado na consola do Raspberry, pois este não coincidia com o *byte* transmitido pelos interruptores para FPGA. A partir de uma frequência de 6 MHz existir uma total dessincronização e o protocolo SPI não conseguia de facto funcionar a tal frequência.

Tabela 4.1: Tabela de resultados para o teste de comunicação no sentido FPGA – Raspberry Pi.

Número de mensagens	Frequência	Número de erros	Percentagem de sucesso
50	1 MHz	0	100%
250		0	
2500		0	
25000		0	
50	2 MHz	0	100%
250		0	
2500		0	
25000		0	
50	3 MHz	0	100%
250		0	
2500		0	
25000		0	
50	4 MHz	0	100%
250		0	
2500		0	
25000		0	
50	5 MHz	8	84%
250		32	87.20%
2500		340	86.40%
25000		3520	85.92%
50	6 MHz	50	0%
250		250	
2500		2500	
25000		25000	

4.2.2 Testes de frequência na comunicação Raspberry Pi para a FPGA

Tendo então “diagnosticado” a frequência máxima permitida pelo Raspberry Pi no uso do protocolo SPI para a transmissão de informação entre o Raspberry e a FPG, foi realizado o teste no outro sentido para confirmar que a FPGA também tinha sucesso de receção a essa frequência. Desta forma e aprofundando o teste chegou-se a perceber qual seria a frequência máxima a que

a FPGA conseguiria manter a eficácia de transmissão e foi realizado o seguinte teste. O Raspberry Pi enviou mensagens para a FPGA sob forma de um contador, portanto se enviar 50 mensagens, a primeira mensagem enviada vai ser o valor 0, de seguida o valor 1, por aí adiante até enviar o valor 49 na última mensagem. A FPGA mostra as mensagens recebidas e forma binária recorrendo a 8 LED incorporados. Este teste vai ser feito com um ***delay* de 0.5 segundos** entre envio de mensagens para que possa ser visível se o contador está a ser recebido de forma correta. No final do envio vai-se registar o *byte* mostrado nos LED da última mensagem recebida.

Este teste foi feito com *delay* de modo que fosse possível observar com clareza a contagem das mensagens através dos LED da FPGA. A FPGA funciona a uma maior frequência que o Raspberry e como podemos observar por este teste, a FPGA manteve a eficácia de receção até a uma frequência de 17.5 MHz (ver tabela 4.2). Depois desses valores é observável alguns erros na receção das mensagens, mas era percetível a contagem, sendo que estes erros apareciam como *shifts* dos bits ou aparecia o código de erro implementado na FPGA. Após a frequência de 21 MHz a quantidade de erros recebidos era muito superior e já não se percebia o que se estava de facto a transmitir.

Tabela 4.2: Tabela de resultados do teste de comunicação no sentido Raspberry Pi - FPGA.

Número de mensagens	Frequência	Resultado final LED	Erro Observável
50	5 MHz	"00110001"	Não
250		"11111001"	
500		"11110011"	
50	10 MHz	"00110001"	Não
250		"11111001"	
500		"11110011"	
50	12.5 MHz	"00110001"	Não
250		"11111001"	
500		"11110011"	
50	15 MHz	"00110001"	Não
250		"11111001"	
50	17.5 MHz	"00110001"	Não
250		"11111001"	
50	20 MHz	"00110001"	Sim, 2/3 vezes
250		"11111001"	Sim, 15/20 vezes
50	21 MHz	"00110001"	Sim, 2/3 vezes
250		"11111001"	Sim, 20/25 vezes
50	22.5 MHz	Código de erro	Sim, não é percetível
250		Código de erro	Sim, não é percetível
50	25 MHz	Código de erro	Sim, sempre código de erro
250		Código de erro	

Tendo em conta os resultados destes testes, os testes realizados posteriormente foram realizados a uma frequência de 4 MHz para que não ocorresse erros quer ao nível do Raspberry a enviar códigos para a FPGA, quer ao nível da FPGA a enviar o pixel pedido para o Raspberry.

4.2.3 Testes no Envio de Imagens

A figura 4.2 contém o código Python do *master* que envia um código de inicialização para a FPGA seguido de diversos pedidos de um pixel RGB (código 0x06) de forma a refazer a imagem. Para cada pedido de um pixel o *master* envia o código recebendo o *byte* que contém três *bits* de vermelho, três *bits* de verde e dois *bits* de azul, portanto tem que se fazer uma conversão para um valor RGB entre 0 e 255 para posteriormente guardar o pixel fazendo um mapeamento da imagem.

```
w, h = 320, 240
data = np.zeros((h, w, 3), dtype=np.uint8)

msgInitial = [0xc7]
[resultInitial] = spi.xfer2(msgInitial)
time.sleep(0.001)

for y in range(h):

    for x in range(w):

        RedValue = 0
        GreenValue = 0
        BlueValue = 0

        msg = [0x06]
        [result] = spi.xfer2(msg)

        BinaryNumber = bin(result)[2:]

        while len(BinaryNumber) < 8:
            BinaryNumber = '0' + BinaryNumber

        if BinaryNumber[0] == '1':
            RedValue = RedValue + 128
        if BinaryNumber[1] == '1':
            RedValue = RedValue + 64
        if BinaryNumber[2] == '1':
            RedValue = RedValue + 32

        if BinaryNumber[3] == '1':
            GreenValue = GreenValue + 128
        if BinaryNumber[4] == '1':
            GreenValue = GreenValue + 64
        if BinaryNumber[5] == '1':
            GreenValue = GreenValue + 32

        if BinaryNumber[6] == '1':
            BlueValue = BlueValue + 128
        if BinaryNumber[7] == '1':
            BlueValue = BlueValue + 64

        data[y, x] = [RedValue, GreenValue, BlueValue]

img = Image.fromarray(data, 'RGB')
img.save('RGBImage.png')
img.show()
```

Figura 4.2: Código implementado no Raspberry Pi que constrói imagem RGB 320x240 através do protocolo SPI com a FPGA.

O código apresentado na figura 4.3 apresenta os três canais de cada pixel. Neste caso a variável Din, de 12 *bits* é divida pelas três componentes, sendo que os primeiros quatro *bits* mais significativos correspondem à cor vermelha, os seguintes quatro à cor verde e os últimos quatro *bits*, os quatro menos significativos estão associados à cor azul. Nesta situação vamos usar os três *bits* mais significativos do canal vermelho, os três *bits* mais significativos do canal verde e os dois *bits* mais significativos do canal azul. Desta forma formamos um *byte* que vai ser transmitido para o Raspberry Pi. Os sinais sR, sG e sB são os sinais que vão gerar o pixel para posteriormente ser mostrado pela FPGA no monitor.

```
196         sR <= Din(11 downto 9) & "00000";
197         sG <= Din(7 downto 5) & "00000";
198         sB <= Din(3 downto 2) & "000000";
199         pixel_to_send <= Din(11 downto 9) & Din(7 downto 5) & Din(3 downto 2);
```

Figura 4.3: Código VHDL que compõe o pixel RGB da imagem capturada.

A imagem capturada pela câmara, é apresentada na figura 4.4. Esta não tem a melhor qualidade devido às especificações da própria câmara 0V7670. Ainda assim, a imagem é nítida e podemos observar um tom esverdeado nas cores mais claras.

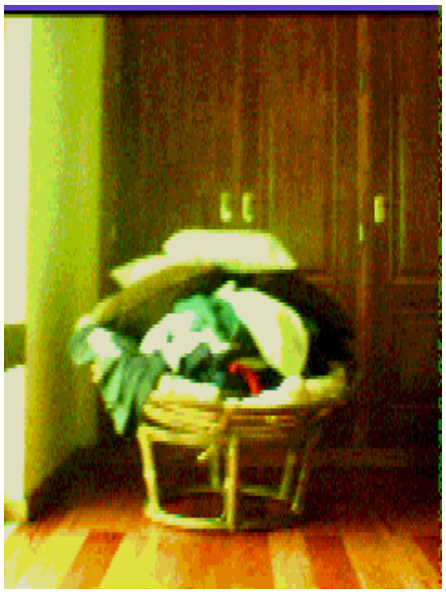

Figura 4.4: Imagem RGB mostrada pelo Raspberry Pi.

4.2.4 Red, Green e Blue Channel

Para se receber apenas uma componente de cor, é necessário utilizar os *bits* associados a cada componente da variável Din, que tem três *bits* correspondendo aos *bits* mais significativos de cada componente, e atribuí-los consoante o código desejado como se mostra na figura 4.5. Neste caso específico o código 0x00 convertia a imagem capturada pela câmara num *red channel*, o código 0x01 convertia num *green channel* e o código 0x02 num *blue channel*.

```
165     if nextColor = "000" then--red channel
166         sR <= Din(11 downto 8) & "0000";
167         sG <= Din(11 downto 8) & "0000";
168         sB <= Din(11 downto 8) & "0000";
169         pixel_to_send <= Din(11 downto 8) & "0000";
170     elsif nextColor = "001" then--green channel
171         sR <= Din(7 downto 4) & "0000";
172         sG <= Din(7 downto 4) & "0000";
173         sB <= Din(7 downto 4) & "0000";
174         pixel_to_send <= Din(7 downto 4) & "0000";
175     elsif nextColor = "010" then--blue channel
176         sR <= Din(3 downto 0) & "0000";
177         sG <= Din(3 downto 0) & "0000";
178         sB <= Din(3 downto 0) & "0000";
179         pixel_to_send <= Din(3 downto 0) & "0000";
```

Figura 4.5: Código VHDL que realiza a atribuição das componentes de modo a implementar a transmissão de um *red*, *green* e *blue channel*.

Podemos observar pela figura 4.6, que a imagem *red channel* se aproxima bastante da imagem *green channel* e que a *blue channel* é acentuadamente mais escura do que o das outras duas componentes, isto deve-se ao facto de a imagem capturada pela câmara ter mais tons no sentido da cor vermelha e verde e pouco influência da componente azul.

Figura 4.6: Imagens mostradas pelo Raspberry Pi (Esquerda – *red channel*; Centro – *green channel*; Direita – *blue channel*).

4.2.5 Imagem em tons de cinzento

Para se converter uma imagem RGB numa imagem em tons de cinzento, somou-se as componentes dando mais peso à componente verde (tem o dobro do peso das outras duas componentes). O resultado é o pixel em tons de cinzento a ser enviado. Na figura 4.7 os sinais sR, sG e sB guardaram o valor do somatório para que seja possível visualizar a imagem transmitida pela FPGA para o monitor por VGA.

```
201         totalR := Din(11 downto 8) & "00";--x4
202         totalG := Din(7 downto 4) & "000";--x8
203         totalB := Din(3 downto 0) & "00";--x4
204         totalGray := to_integer(unsigned(totalR)) + to_integer(unsigned(totalG)) + to_integer(unsigned(totalB));
234         sR <= std_logic_vector(to_unsigned(totalGray,8));
235         sG <= std_logic_vector(to_unsigned(totalGray,8));
236         sB <= std_logic_vector(to_unsigned(totalGray,8));
237         pixel_to_send <= std_logic_vector(to_unsigned(totalGray,8));
```

Figura 4.7: Código VHDL que implementa a lógica de converter um pixel RGB num pixel em tons de cinzento.

Na figura 4.8 podemos ver o resultado do teste descrito. Claramente encontramos diferentes intensidades ao longo da imagem que se distingue zonas mais escuras com menor intensidade e zonas mais brancas com maior intensidade no espectro de tons de cinzento.

Figura 4.8: Imagem em tons de cinzento.

4.2.6 Imagem binarizada

Para testar a binarização da imagem em tons de cinzento, foi necessário enviar o *threshold* desejado antes de pedir os pixéis. Caso esse valor do pixel seja superior ao *threshold* é enviado um pixel branco, caso seja menor ou igual é enviado um pixel preto como demostrado no código VHDL da figura 4.9.

```
224         if binarize = '1' then
225             if totalGray > threshold then
226                 totalGray := 255;
227             else
228                 totalGray := 0;
229             end if;
```

Figura 4.9: Código VHDL que executa a lógica para a binarização.

Começou-se por enviar uma imagem se com o *threshold* 127, que é o valor de omissão definido no código VHDL. O *master* envia um código em que um dos bits irá definir se a imagem em tons de cinzento será binarizada ou não (o quarto bit menos significativo).

Observando a figura 4.10 e comparando as duas imagens, podemos concluir que a binarização foi realizada com sucesso e que as zonas mais claras da imagem foram substituídas por um pixel branco e as zonas mais escuras por um pixel preto.

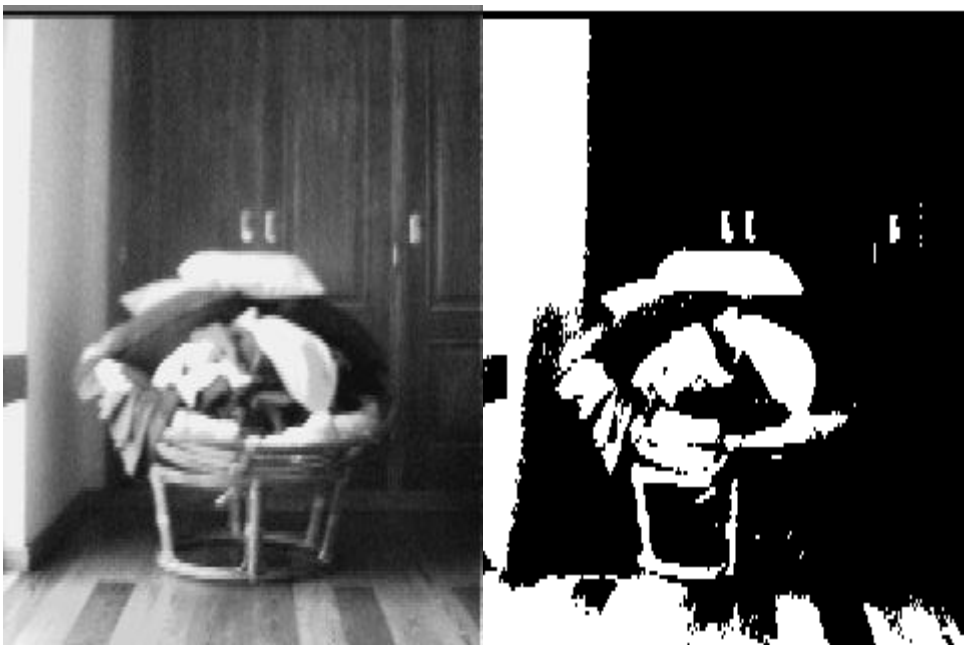

Figura 4.10: Imagens mostradas pelo Raspberry Pi (Esquerda – Imagem em tons de cinzento; Direita – Imagem binarizada com *threshold* de 127).

Nos testes seguintes alterou-se o *threshold* antes de enviar a imagem (para os valores 64 e 192). Como referido anteriormente, uma das implementações foi realizar a binarização dando a possibilidade de o *master* escolher qual o valor do *threshold* utilizado na execução da binarização. Esta lógica funciona da mesma maneira, pois é uma binarização, só que o *master* ao enviar o código de inicialização vai enviar um código que indica que é uma binarização personalizada e irá também enviar o *threshold* desejado. Assim sendo volta a invocar uma transmissão e passa o valor do *threshold* no *byte*.

Observando a figura 4.11, podemos concluir que de facto quando menor o *threshold* mais pixéis brancos tem a imagem, pois diminuindo o *threshold* obviamente temos mais pixéis com valor acima deste. O inverso acontece caso o *threshold* aumente e a imagem fica com mais pixéis pretos.

Figura 4.11: Imagens mostradas pelo Raspberry Pi (Esquerda – Imagem binarizada com *threshold* de 64; Centro - Imagem binarizada com *threshold* de 127; Direita – Imagem binarizada com *threshold* de 192).

4.2.7 Negativo

Para testar o negativo, onde a lógica é inverter o valor do pixel, ou seja, subtrair ao valor máximo (pixel branco, valor 255) o próprio valor do pixel em questão como mostra a figura 4.12. Novamente esta função foi implementada sobre uma imagem em tons de cinzento e como tal existe uma variável associada a um *bit* do comando enviado pelo *master* que faz essa distinção entre a opção para processar o pixel para negativo ou não (foi usado o quinto bit menos significativo). Na sequência do que foi feito, os sinais sR, sG e sB são sinais que posteriormente vão refletir a imagem negativa no monitor por VGA através da FPGA.

```
231         if negative = '1' then
232             totalGray := 255 - totalGray;
233         end if;
234         sR <= std_logic_vector(to_unsigned(totalGray,8));
235         sG <= std_logic_vector(to_unsigned(totalGray,8));
236         sB <= std_logic_vector(to_unsigned(totalGray,8));
237         pixel_to_send <= std_logic_vector(to_unsigned(totalGray,8));
238     end if;
239 end process;
```

Figura 4.12: Código VHDL que implementa o processamento do pixel em tons de cinzento para negativo.

Podemos visualizar na figura 4.13 o efeito do negativo numa imagem. Os tons mais claros mudam para tons escuros e vice-versa, já em relação ao negativo de uma binarização, como apenas temos dois possíveis valores (branco ou preto) a troca é bastante fácil, visto que tudo o que é branco passa a ser preto e os pixéis pretos torna-se pixéis brancos.

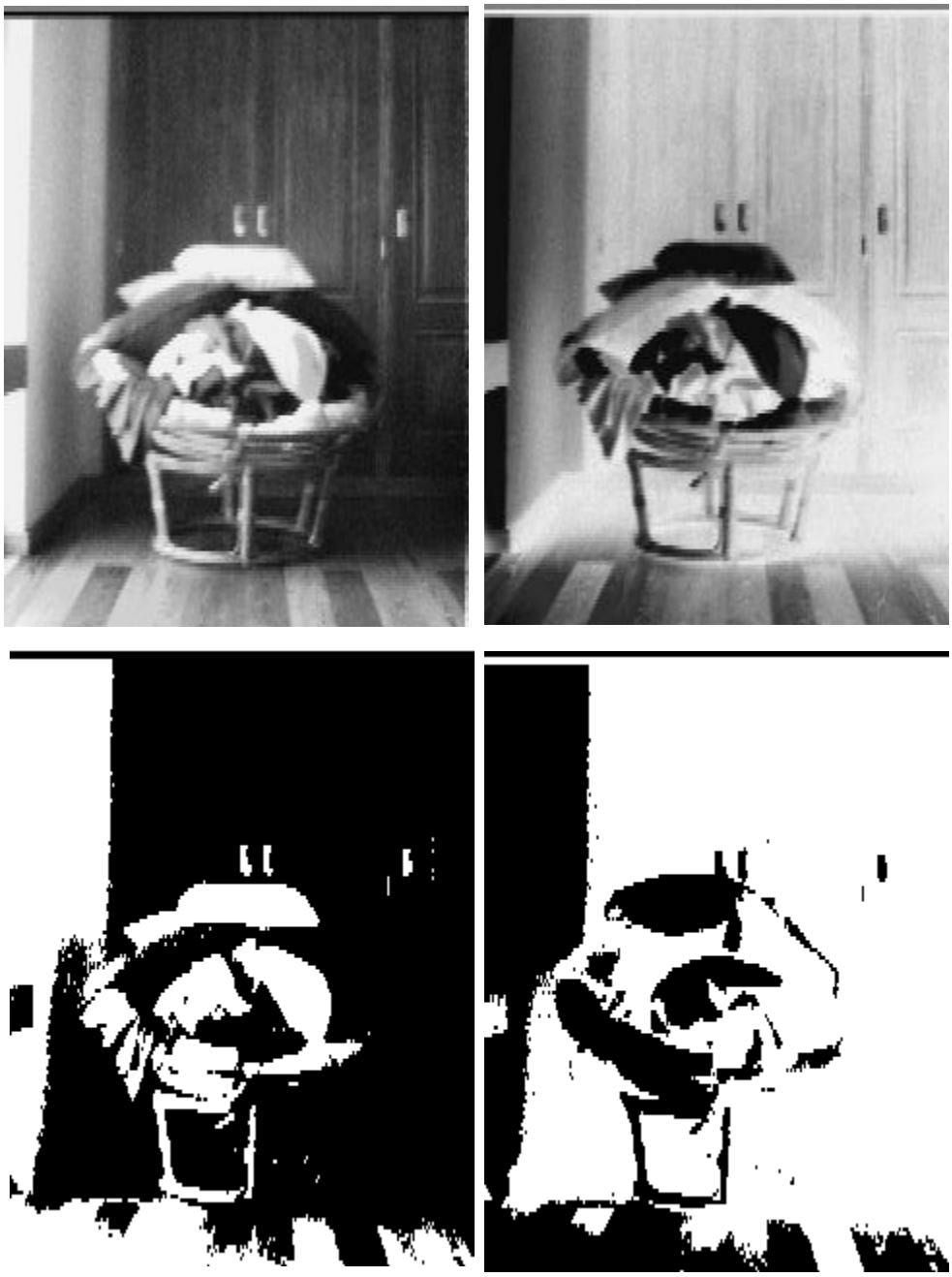

Figura 4.13: Imagens obtidas no Raspberry Pi (Canto superior esquerdo – Imagem em tons de cizento; Canto superior esquerdo – Negativo da imagem em tons de cinzento; Canto inferior esquerdo – Imagem binarizada com *threshold default*; Canto inferior direito – Negativo da imagem binarizada).

4.2.8 Modificar o brilho da imagem

Para modificar o brilho da imagem utiliza-se uns registos que guardam um valor entre 0 e 255, que são inicializados com o valor zero, mas caso o master envie um respetivo código, esses valores são alterados o que vão influenciar o brilho da imagem em tons de cinzento.

Podemos observar pela figura 4.14, que ao somar ou subtrair o valor do brilho, tem que se limitar o valor mínimo e máximo, 0 e 255 respetivamente, para que não aconteça a situação de o valor ultrapassar o valor máximo e ainda adicionar o resto a partir do valor mínimo.

```
214         if totalGray + increaseBrightness > 255 then
215             totalGray := 255;
216         else
217             totalGray := totalGray + increaseBrightness;
218         end if;
219         if totalGray - decreaseBrightness < 0 then
220             totalGray := 0;
221         else
222             totalGray := totalGray - decreaseBrightness;
223         end if;
```

Figura 4.14: Código VHDL que aumenta ou diminui o valor do pixel transmitido ao Raspberry Pi.

Observando a figura 4.15, é fácil de ver que a imagem da esquerda tem bastante mais brilho que a imagem da direita. Podemos ver que o lado esquerdo da imagem com brilho acrescentado está muito perto de branco, o que faz reparar que é importante limitar o valor máximo, pois se não tivesse limitado poderia aparecer pixéis perto de preto naquela zona.

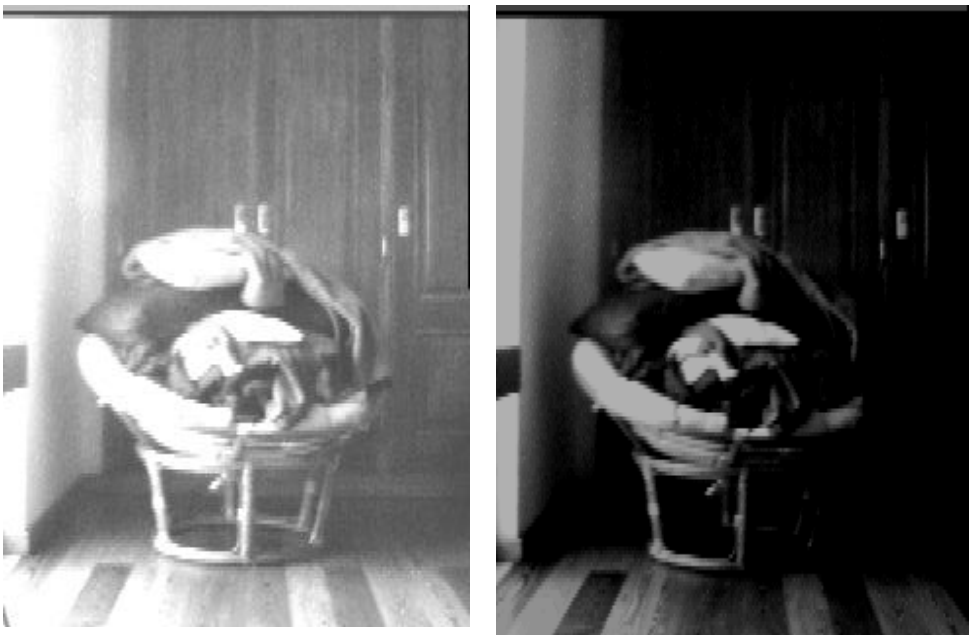

Figura 4.15: Imagens gravadas no Raspberry (Esquerda – Imagem com brilho acrescentado com valor 64; Direita – Imagem com brilho decrementado com valor 64).

4.2.9 Modificar o contraste da imagem

É possível mudar o contraste da imagem multiplicando ou dividindo o valor de cada pixel em tons de cinzento por 2. Também ao alterar o contraste, tal como na alteração do brilho é necessário garantir que não se ultrapassa o valor máximo de 255, para que não passe o valor correspondente ao pixel branco e se torne um pixel preto. Na figura 4.16 apresenta-se o código VHDL correspondente.

```
205         if contrast = 1 then
206             if totalGray * 2 > 255 then
207                 totalGray := 255;
208             else
209                 totalGray := totalGray * 2;
210             end if;
211         elsif contrast = 2 then
212             totalGray := totalGray / 2;
213         end if;
```

Figura 4.16: Código VHDL que altera o valor do pixel modificando o contraste da imagem.

Podemos visualizar na figura 4.17, que a imagem esquerda tem muito mais claridade que a imagem à direita. Se tivesse aumentado a variável que indicava o quanto iria multiplicar ou dividir, a diferença de claridade entre as duas imagens seria maior.

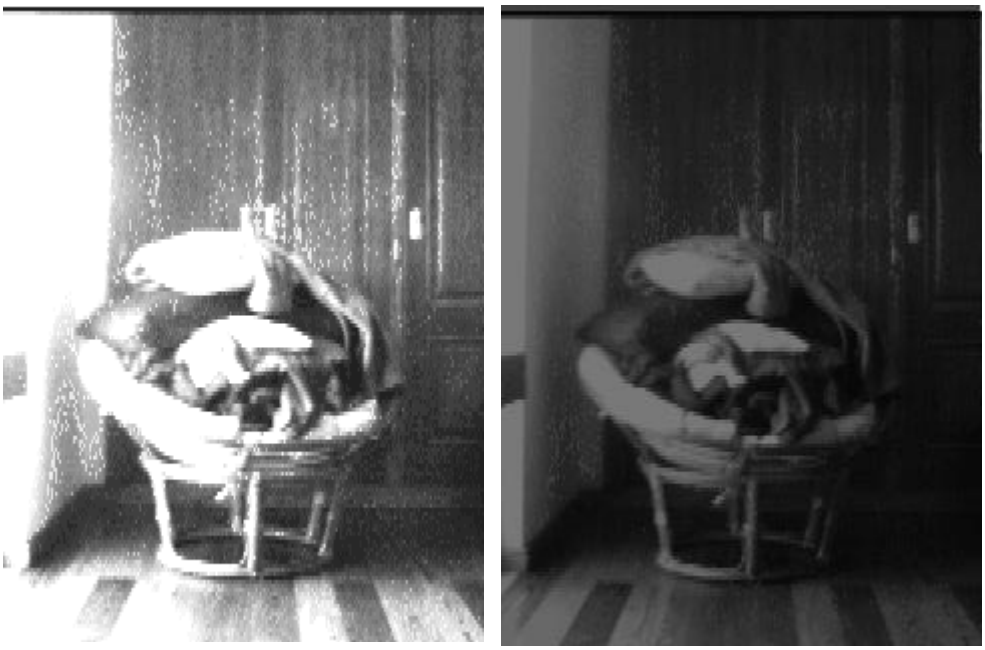

Figura 4.17: Imagens adquiridas no Raspberry Pi (Esquerda – Imagem com contraste multiplicando por dois; Direita – Imagem com contraste dividindo por dois).

4.3 Recursos Utilizados

Neste projeto a imagem capturada pela câmara com uma resolução de 640x480 pixéis é guardada parcialmente na *BlockRAM* (BRAM) da FPGA. Isto porque a BRAM é insuficiente para guardar esta quantidade de pixéis. Portanto a alternativa foi invés de guardar a imagem completa, apenas se guarda um pixel a cada quatro e assim já tem espaço para manter essa imagem reduzida em memória. Mesmo com este pequeno truque a ocupação da BRAM mantêm-se nos 89%, como mostra a figura 4.18. O resto dos recursos estão muito abaixo do máximo dsiponível.

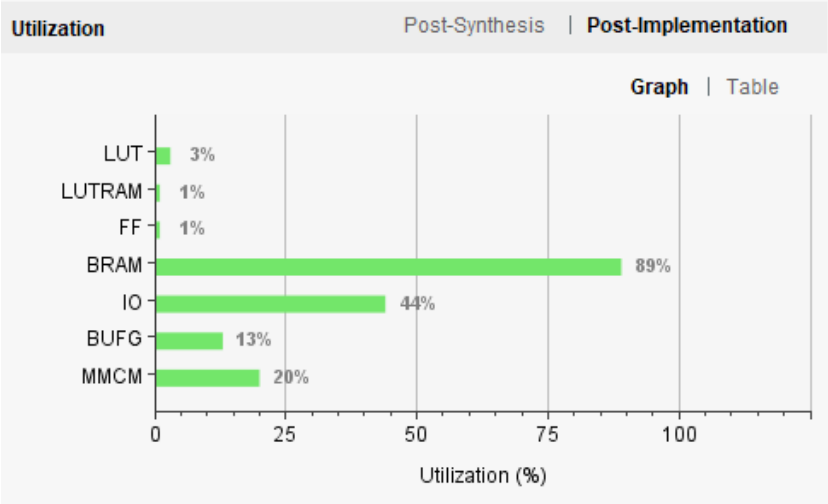

Figura 4.18: Utilização dos recursos da Basys 3 neste projeto.

5

5. Conclusões e Trabalho Futuro

Neste último capítulo apresentam-se as conclusões do trabalho desenvolvido na presente dissertação. Também se referem algumas sugestões para desenvolvimentos futuros e a utilização de plataforma *hardware* com maior capacidade de memória e de computador mais rápido.

5.1 Conclusões

Atualmente o assunto de processamento de imagens é quase inevitável para o nosso dia a dia. Isto porque tudo o que envolve captura de imagem vai necessitar de processamento. O processamento está cada vez mais rápido, mas continua a ser pouco acessível ao nível do *hardware*, daí o interesse em plataformas heterogéneas para captar e processar imagens.

A solução proposta tens duas vertentes, uma parte em *hardware* e uma parte em *software*, assim sendo dividindo tarefas de processamento, executando as menos complexas em *hardware* e as mais difíceis de implementar em *software*.

Este sistema foi pensado e desenhado de forma a poder disponibilizar um projeto funcional e modular que possa servir como plataforma de prototipagem e aprendizagem para utilizadores menos experientes. Sendo ainda de baixo custo. Este projeto poderá ser aumentado no sentido de disponibilizar métodos adicionais para processamento de imagens. Os recursos da FPGA utilizada estão longe do seu limite, com exceção da memória BRAM.

O protocolo de comunicação SPI permitiu uma comunicação fiável entre o Raspberry Pi e a FPGA. Foi possível verificar que no sentido Raspberry FPGA, era possível atingir uma velocidade de 17.5MHz e no sentido contrário 4MHz.

5.2 Trabalho Futuro

Como proposta de trabalho futuro, sugere-se implementar métodos adicionais de pré-processamento, tais como projeções horizontais e verticais, adequadas por exemplo para segmentar imagens.

Seria de interesse implementar o mesmo tipo de sistema heterogéneo com um computador mais rápido para permitir uma maior taxa de transferência de dados através de SPI. Assim fosse, o computador podia receber da FPGA a imagem em menos tempo.

Relativamente à ocupação da BRAM da FPGA, sugere-se então uma FPGA com uma maior *BlockRAM*. Desta forma seria possível ter toda a imagem guardada/disponível em memória, não sendo necessário perder informação.

Referências

[1] Yousif Mohamed Y. Abdallah and Tariq Alqahtani, “Research in Medical Imaging Using Image Processing Techniques”, IntechOpen, 2019, DOI: 10.5772/intechopen.84360.

[2] T. Carrasqueira, F. Moutinho, and R. Campos-rebelo, “FPGA in image processing,” *REC’2019 – XV Jornadas sobre Sist. Reconfiguráveis*, 2019.

[3] E. Suganya, M. Karthiga, “In Internet of Things in Biomedical Engineering”, Chapter 5 – “IoT In Agriculture Investigation on Plant Diseases and Nutrient Level Using Image Analysis Techniques”, 2019.

[4] D. Sanjay, P. Rajesh Kumar and T. Satya Savithri, "Fuzzy Control for Person Follower FPGA based Robotic System", Indian Journal of Science and Technology 8(23), 2015.

[5] “MATLAB - MathWorks - MATLAB & Simulink” [Online], Available: https://www.mathworks.com/products/matlab.html. [Accessed: 25-Aug-2020]

[6] O. Marques, “Practical Image and Video Processing Using MATLAB”, Wiley – IEEE Press, 2011, ISBN: 978-1-118-09347-4.

[7] S. Matuska, R. Hudec, and M. Benco, “The comparison of CPU time consumption for image processing algorithm in Matlab and OpenCV,” 2012 ELEKTRO, 2012, DOI: 10.1109/ELEKTRO.2012.6225575.

[8] “OpenCV” [Online], Available: https://opencv.org/. [Accessed: 25-Aug-2020]

[9] Bo Li, Aleksandar Jevtic, Ulrik Soderstrom, Shafiq Ur Réhman, Haibo Li, “Fast Edge Detection by Center of Mass”, 1st IEEE/IIAE International Conference on Intelligent Systems and Image Processing (ICISIP2013), 2013

[10] Priyabrata Biswas, “Introduction to FPGA and its Architecture”, 2019, [Online], Available: https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c. [Accessed: 11-Aug-2020]

[11] T. Chisholm, R. Lins and S. Givigi, "FPGA-Based Design for Real-Time Crack Detection Based on Particle Filter," in IEEE Transactions on Industrial Informatics, vol. 16, no. 9, pp. 5703-5711, Sept. 2020, doi: 10.1109/TII.2019.2950255.

[12] A. HajiRassouliha, A. J. Taberner, M. P. Nash e P. M. Nielsen. “Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms”. Em: Signal Processing: Image Communication 68. July (2018), pp. 101–119.

[13] J. Fowers, G. Brown, P. Cooke e G. Stitt. “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications”. Em: Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. 2012, pp. 47–56.

[14] S. Asano, T. Maruyama e Y. Yamaguchi. “Performance comparison of FPGA, GPU and CPU in image processing”. Em: FPL 09: 19th International Conference on Field Programmable Logic and Applications (2009), pp. 126–131. doi: 10.1109/FPL.2009.5272532.

[15] “Vivado Design Suite - Xilinx” [Online], Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed: 25-Aug-2020].

[16] “VHDL” [Online], Available: https://mecawiki.fandom.com/pt-br/wiki/VHDL. [Accessed: 25-Aug-2020].

[17] "IEEE Standard for VHDL Language Reference Manual," in IEEE Std 1076-2019, vol., no., pp.1-673, 23 Dec. 2019, doi: 10.1109/IEEESTD.2019.8938196.

[18] "IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[19] D. J. Smith, "VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C," 33rd Design Automation Conference Proceedings, 1996, Las Vegas, NV, USA, 1996, pp. 771-776, doi: 10.1109/DAC.1996.545676.

[20] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones, "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels", 2019 IEEE International Conference on Embedded Software and Systems (ICESS), 2019.

[21] “Xilinx - Adaptable. Intelligent.” [Online], Available: https://www.xilinx.com/. [Accessed: 25-Aug-2020].

[22] “Intel® FPGAs and Programmable Devices - Intel® FPGA” [Online], Available: https://www.intel.com/content/www/us/en/products/programmable.html. [Accessed: 25-Aug-2020].

[23] “Zybo Z7 - Digilent Reference” [Online], Available: https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/start. [Accessed: 25-Aug-2020].

[24] “How to select the best FPGA” [Online], Available: https://numato.com/blog/how-to-select-the-best-fpga-for-your-application. [Accessed: 25-Aug-2020].

[25] Johny Paul, “Image Processing on Heterogeneous Multiprocessor System-on-Chip using Resource-aware Programming”, Technical University Munich, Germany, 2017.

[26] Zalak Dave, Shivank Dhote, Pranav Charjan, Jonathan Joshi and Ganesh Gore. “Reconfigurable Image Processor using an FPGA-Raspberry pi Interface”. IJCA Proceedings on International Conference on Computer Technology ICCT 2015(5):11-15, September 2015.

[27] Kumar Santosh, Shah Madhu, Singh Arjun, “FPGA – Raspberry pi Interface for low cost IoT based image processing”, Invertis Journal of Science & Technology, 2017, DOI: 10.5958/2454-762X.2017.00034.8.

[28] A. Rupani, P. Whig, G. Sujediya and P. Vyas, "A robust technique for image processing based on interfacing of Raspberry-Pi and FPGA using IoT," 2017 International Conference on Computer, Communications and Electronics (Comptelix), Jaipur, India, 2017, pp. 350-353, doi: 10.1109/COMPTELIX.2017.8003992.

[29] “AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite” [Online], Available: https://developer.arm.com/documentation/ihi0022/e/. [Accessed: 25-Aug-2020].

[30] “Basics of the SPI Communication Protocol” [Online], Available: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol. [Accessed: 25-Aug-2020].

[31] S. Michalak, "Raspberry Pi as a measurement system control unit," 2014 International Conference on Signals and Electronic Systems (ICSES), Poznan, Poland, 2014, pp. 1-4, doi: 10.1109/ICSES.2014.6948735.

[32] Pang A., Membrey P. “Two-Way Communications with Your Raspberry Pi: SPI”, In: Beginning FPGA: Programming Metal. Apress, Berkeley, CA. 2017, https://doi.org/10.1007/978-1-4302-6248-0_15.

[33] Haissam Hajjar, Hussein Mourad, “Implementation of an FPGA - Raspberry Pi SPI Connection”, CENICS 2019: The Twelfth International Conference on Advances in Circuits, Electronics and Micro-electronics, 2019.

[34] “SPI e I2C” [Online], Available: https://paginas.fe.up.pt/~hsm/docencia/comp/spi-e-i2c. [Accessed: 25-Aug-2020].

[35] “Basics of the I2C Communication Protocol” [Online], Available: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol. [Accessed: 25-Aug-2020].

[36] B. Eswari, N. Ponmagal, K. Preethi and S. G. Sreejeesh, "Implementation of I2C master bus controller on FPGA," 2013 International Conference on Communication and Signal Processing, Melmaruvathur, India, 2013, pp. 1113-1116, doi: 10.1109/iccsp.2013.6577229.

[37] “OV7670/OV7671 CMOS VGA (640x480) CameraChip with OmniPixel Technology” [Online], Available: https://www.voti.nl/docs/OV7670.pdf. [Accessed: 11-Aug-2020].

[38] “Diligent Basys 3 Reference” [Online], Available: https://reference.digilentinc.com/basys3/refmanual. [Accessed: 23-Aug-2020].

[39] “Basys 3 FPGA OV7670 Camera” [Online], Available: https://www.fpga4student.com/2018/08/basys-3-fpga-ov7670-camera.html. [Accessed: 25-Aug-2020].

[40] “Raspberry Pi GPIO Datasheet” [Online], Available: https://www.raspberrypi.org/documentation/usage/gpio. [Accessed: 25-Aug-2020].

[41] “spidev · PyPI” [Online], Available: https://pypi.org/project/spidev/. [Accessed: 25-Aug-20].